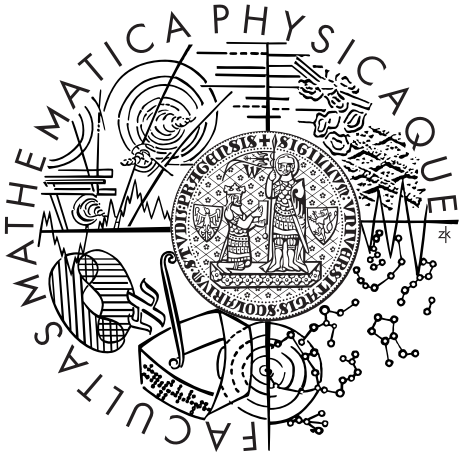
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Karel Jakubec

Management of Undo/Redo Operations in Complex Environments

Department of Software Engineering

Supervisor of the master thesis::	RNDr. Irena Mlýnková, Ph.D.
Study program:	Computer science
Specialization:	Software engineering

Prague 2012

I wish to express my gratitude and appreciation to my supervisor, RNDr. Irena Mlýnková, Ph.D. for her thoughtful guidance, valuable suggestions during many discussions and her kind supervision of this diploma thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Management of Undo/Redo Operations in Complex Environments

Autor: Karel Jakubec

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.

Abstrakt: Během posledních třiceti let bylo prezentováno několik nelineárních algoritmů, jejichž snahou bylo zlepšit řízení operací undo a redo (zpět a vpřed). Takřka žádný však nenavrhoval, jak řídit tyto operace v prostředí s několika pracovními plochami, kde jednotlivé uživatelské akce na různých pracovních plochách mohou být navzájem provázány.

Cílem této práce je vyvinout nový algoritmus, který by umožnil uživateli vzít zpět jakoukoli (nikoliv pouze poslední) akci a to právě v prostředí, kde uživatel pracuje na vícero pracovních plochách současně. Algoritmus se musí vypořádat se zavislostmi mezi jednotlivými akcemi a korektně je vyřešit tak, aby dokument zůstal ve stabilním stavu. Výsledky jsou prezentovány ve frameworku DaemonX.

Klíčová slova: řízení, vpřed, zpět, vícero pracovních ploch

Title: Management of Undo/Redo Operations in Complex Environments

Author: Karel Jakubec

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.

Abstract: During past thirty years, several models for non-linear undo models have been presented, but almost none solves undoing and redoing actions in environments, where multiple history buffers are involved and when there are causal dependencies among separate actions.

This thesis focuses on developing a new model, which allows a user to select any action from any history buffer. The key part of the model is a smart command design and undo manager, which searches for dependencies and offers possible solutions to the user. The results are presented in the context of the DaemonX framework.

Keywords: selective undo, selective redo, multiple workspaces

Contents

List of Figures

1. Introduction

An undo and redo functionality is today a common feature of various interactive applications and systems. Many users can not imagine an editor without this support. It is essential for new users, who can explore various features of an application without being afraid of destroying the data. Accidental mistakes can be fixed in magnitude of seconds, just by pressing the undo button.

Most of today's applications use a very simple model - all user's actions are stored in a history stack. Only the most recent action may be undone and this is usually done by performing an inverse operation. Such an action is marked as undone and the user may redo it or undo the previous action. When a new action is invoked, all actions to redo are discarded.

This simple model, commonly known as *linear undo model*, is sufficient in many applications. Its simplicity is a big advantage - most of the users are familiar with it and they expect such behavior. The result of undo or redo operation may be easily predicted and also the physical implementation is not very complex using *command* design pattern [10]. This pattern divides user's actions to a series of discrete steps, which can be later reversed (undone).

But there are environments, where this approach may not be sufficient to fulfill user's needs [6]. If there are non-trivial dependencies among actions - for instance when one document is being edited simultaneously on multiple workspaces or one object is being modified from different perspectives - the simple linear undo model may not be user-friendly.

The other challenging task is to implement a model, which allows a user to undo any action at any time - so-called *selective undo*. This feature can greatly save user's time, but its results tend to be unpredictable and it is hard to implement it even in simple environments.

1.1 Aims of this Thesis

This thesis aims to explore possibility of undo/redo management in such a complex environment and create an algorithm, which allows the user to undo any command at any time in this environment. The algorithm will have to deal with documents spread over several workspaces and it will have to successfully manage correct undoing of actions, which are dependent on each other.

The algorithm will be then implemented in the DaemonX [12] - a framework for data modeling developed at the Faculty of Mathematics and Physics of the Charles University in Prague. Its main purpose is to provide a set of tools to model a modeling language (also known as *meta-modeling*), mechanism for data propagation among various modeling languages and common runtime environment. Each modeling language can be then implemented as a solitaire plug-in which runs in the framework's common runtime environment. One document in DaemonX may consists of several diagrams, each possibly in a different modeling language. This fact has brought unexpected difficulties during implementation of undo/redo functionality, because linear undo model was slow and results were not sufficient for effective work with the framework.

The motivation for the thesis is not only to solve this problem in the DaemonX

framework, but it is a need for general algorithm, which would be able to provide selective undo functionality in non-trivial environments.

1.2 Structure of the Thesis

Chapter 2 discusses undo and redo operations generally. It briefly goes through history and gives a few examples of not so obvious usage of these operations. It points out in which situation undo/redo is meaningful, when its presence is important for the user and what it really brings to the user.

Chapter 3 defines basic terms and properties, which are used later in the thesis.

Chapter 4 defines a common undo model and summarizes the current state of the art. Several real-world undo models based on the command design pattern are presented with an emphasis on non-linear undo models, which are much more challenging, than simple linear undo.

Chapter 5 analyzes issues connected with selective undo and submits several solutions how these issues can be solved or at least avoided.

Chapter 6 presents three algorithms - Extended linear undo model, Cascade selective undo model and the combined undo model - which are capable to maintain undo/redo management in environments with multiple workspaces. The last two of them also offer the possibility of selective undo.

Chapter 7 presents the DaemonX framework, shows real-world implementation of the Combined undo model and presents achieved results.

Chapter 8 summarizes the content of the thesis and suggests possible future work.

2. Overview of Undo and Redo Operations

2.1 History of Undo and Redo Operations

The operation of *undo* is as old as the computer itself. Even the ENIAC, which is considered to be the first general-purpose electronic computer [7], had a mechanism, how to get back to previous points along the solution of a nonlinear differential equation. It created a set of system checkpoints and also provided a mechanism, how to get back to these checkpoints [13].

In those days, the undo (and redo) had a bit different purpose than today. The machine time was far more expensive and so programmers' mistakes also cost much more. The undo in this form was more likely a debugger, which could speed up the programming, but it meant one important thing - the computer is to a certain extent tolerable to operators' (users') mistakes.

The other old reference to an undo operation dates back to 1976. The document *Behavioral issues in the use of interactive systems* [15] discusses, how should a machine interact with the user. The scope of the document is very wide, but there is one important idea: "It would be quite useful to permit users to 'take back' at least the immediately preceding command (by issuing some special 'undo' command)." In this case, the undo operation is targeted to the common users of system and its purpose is to make computer usage easier.

2.2 Undo and Redo Today

An implementation of undo and redo operation can be today found almost in each application, where user's task is to create a document[1] using various functions of application. But there are also other scenarios, which are not so obvious, probably because no "undo" and "redo" buttons are present:

- Undelete a file on a filesystem. This example may have 2 meanings:
  1. Many of today's operating systems, like Microsoft Windows or some distributions of GNU/Linux, support a "trash bin", which is a special folder to which all files, which the user deletes, are moved. Later on, these files may be recovered back to their original state. This folder is a special kind of history buffer. The user typically has to explicitly call action "empty the bin" which deletes the content of the bin (and the user is asked, whether s/he really wants to do it).
  2. An attempt to recover a file from a filesystem, which was lost due to previous deletion. Deletion of a file does not necessary mean that file is lost and cannot be recovered. Many filesystems delete only the information that a file exists but the content of file is still present until it is finally rewritten by another file. Using special tools, files which

[1] The word "document" means any user's work, which is built up using the application. It can be for example a text file, bitmap image, uml project,...

are still not rewritten can be found and recovered and some operating systems (e.g. BSD Unix [1]) even support a system call to do this. Despite the fact, that there is no explicit history buffer, the user is able to undo some of the latest operations.

- Some filesystems have a so-called "journal", which can be seen as a log of actions, which modify the filesystem. The information about these actions are firstly written into a journal and after this, actions are performed on the filesystem itself. When there is a crash during the action performance, it is possible to recover the filesystem into state before the action was performed and filesystem remains in a consistent state. So the journal serves as a history buffer; it holds necessary information to undo the operation.

- When a user changes some vital properties (screen resolution, video output, device driver to use) of an application or the operating system, it may happen that the computer becomes unaccessible. In these situations the user often has to confirm these changes once more after they are applied - if the changes are not confirmed until specific timeout runs out, the original state is restored. Consequently, in this case history buffer has only one entry.

These examples illustrate, that undo and redo operations are not present only in various editors but the user may encounter these operations even during work with the operating system itself and he/she may not even notice it. However, the main domain for undo and redo implementation is still various document editing applications.

2.3 Importance of Undo/Redo Operations

The examples in the previous section together with a typical usage of undo/redo can briefly denote the importance of undo and redo operations in today's systems and applications. The main advantage for the user is the easy recovery from unintentional state of her/his document or system.

The simple fact, that a user is able to undo consequences of his/her action brings:

- Exploration of the application by trying different functions ("What happens when I push this button? "). Each change can be taken back, so the user is less hesitant to try powerful and perhaps unfamiliar operations.

- Unintentional mistakes can be solved quickly (a cat walks over a keyboard and the document is almost lost).

- Possibility to experiment with document format, style and layout ("Take a look, how the text will look with this font.")

3. Definitions

In this chapter some basic terms are defined. The definitions are general and they will be defined more precisely for each scenario later.

3.1 Constructs and Views

For the purpose of this thesis we suppose, that the user works with objects called *constructs*. It is a general object, product of user's work. Each has a unique identifier (*id*), set of *properties* and *views*.

Property may be considered as a simple (key, value) pair.

View is a visualization of the construct to the user. One construct may have several views, each showing the construct from a different perspective. In many situations, one view is sufficient, but there are situations, when a different view of the properties may be useful. The view is usually used for changing the data properties of the construct - it may provide various widgets used by the user. This principle is more deeply described by architecture pattern Model-View-Controller [16].

Sometimes it can be useful to introduce a special property with key *type*. All constructs with the same value of the type property share also the set of keys of all other properties. We say that such a constructs are of the same type. Although it is only syntactic sugar, it simplifies environments where constructs are used for data description or directly as a data storage.

Example 1. *Construct can be:*

- *A letter in a text document. One letter is a construct and the user creates new constructs by typing on the keyboard. Properties of such a constructs are for example the ASCII value, position in a text, font,.... Product of user's work is then a text document, which is a set of constructs.*
- *Any object of UML modeling language. In the editor, which uses UML modeling language[11], any object (UML class, UML relation) can be considered as a construct.*
- *A shape in an image editor. Simple image editor can represent all objects on the screen as a constructs. Properties can be dimensions, color,...*

Construct with several views is described in Figure 3.1.

3.2 Construct Pool and Workspace

All *constructs* created during work reside in a data structure called *construct pool*. This data structure can be implemented in various ways (array, linked list, hashmap,...) and it is up to the programmer of the system, which implementation suits best to his/her needs. One system may also have more than one construct pool.

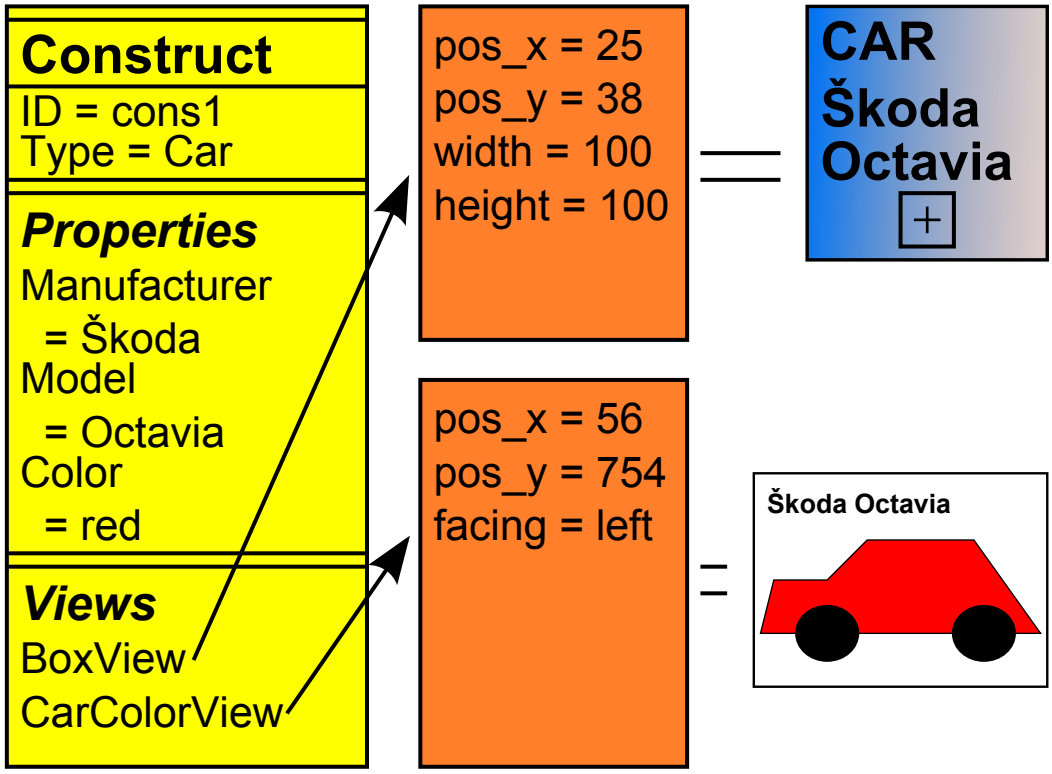


Figure 3.1: Example of construct with two views

One application may also have several *workspaces*. A workspace is a place, where the user performs his/her modifications of constructs. Various workspaces may share one construct pool.

Example 2. *For our purposes, the workspaces may be:*

- *In a text editor, each opened document can be considered as a separate workspace with own construct pool.*
- *In a graphical editor, each image is a separate workspace.*
- *In a designer of diagrams, each diagram can be a separate workspace.*

3.3 Commands

Undo or redo of an action would not be possible, if there was not a mechanism how to delimit and pick exactly one action. In other words, the whole user's work should be a series of discreet steps. These steps are called *commands* and they follow the *command* desing-pattern [10]. Each command is a separate unit of action, which can be either executed or undone and stored in a data structure for later use.

A command represents one action from user's perspective, so it is undivisible for the user. If the user wants to undo a command, he/she must undo the whole command or nothing. But from application perspective the command does not have to be undivisible. Consider situation depicted in Figure 3.2. It shows

simple diagram in UML modeling language [11] with three classes (boxes) and two connections (lines with arrows). The user decides, that there is no need for a common superclass Vehicle and so the class is destroyed. Destruction of this class also means destruction of both generalizations, leading from inherited classes to the Vehicle class. For the user, this is one action - he/she only cares about class destruction and dependencies of other constructs should be solved automatically. Also the undo should be performed in the way, that not only class Vehicle but also all its connections will be restored. But for the application, situation is not so simple - it must destroy three objects, which are three actions from its perspective.

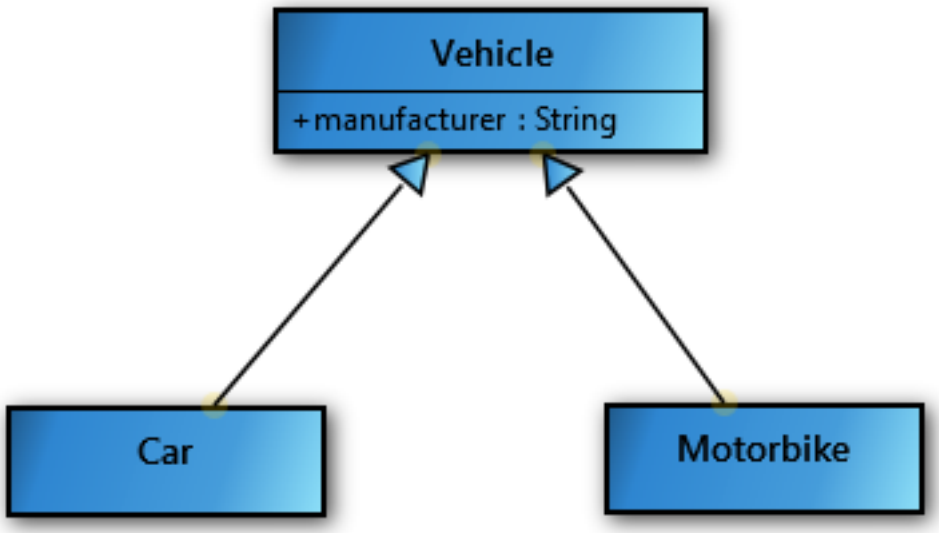


Figure 3.2: Three uml classes with connections

For this reason a so-called *atomic command* is introduced. An atomic command is an atomic action from application part of view and typically it is used to create, destroy or modify one construct or its property. "Composite commands" may then serve as containers for *atomic commands*.

3.3.1 Commands Ordering

The ordering of commands enables to distinguish in which order commands should be undone or redone. Ordering can be local (only one history buffer is involved, often implicit by commands' position in the stack) or global (among all commands in application).

Command C1 is older than command C2 if and only if C1 is predecessor of C2. It means, that C1 was originally executed before C2 was executed.

Command C1 is younger than command C2 if and only if it is not older.

Since equality is not defined, it should be always possible to determine, which command is younger.

3.3.2 Extending Command Structure

By definition, the structure of a command does not carry much information for algorithms, which have to deal with it. To enable some interesting features of undo models, command structure can be extended:

- A simple incremental counter can be added to the command structure. The value of this field is "global" among all commands in the environment. The first command has this number set to zero and the value for each subsequent command is computed as a value of this variable in previous command incremented by 1. From the value of this counter, the chronological order of two commands can be simply determined, even if commands do not share same stack.
- Each command can carry the information about the construct(s) it affects. This is useful, when causal dependencies among various commands need to be taken into account.
- Special redo method. A common command has typically two methods, which actually perform some changes on the model - execute() and undo(), but in some cases, it is good to distinguish between the first execution and the subsequent executions.

3.4 History Logging

There are generally two ways, how the history may be logged:

- The history is logged as a series of document states and a model describes the way, how to move from one state to another. [13]
- The history is logged as a series of operations (similar to commands) and undo is performed as appending an inverse operation to the history buffer. [17]

The first one is memory consuming (each state of document is saved) and because there is generally no system of dependencies among various states, it us harder to support more advanced features of undo models like selective undo.

For this reason, this thesis deals only with the second choice - the executed commands are stored in a data structure called *history buffer*. Implementation of history buffer depends on the used undo model, but in most of today's applications it is a simple LIFO[1] container.

[1] Last In First Out

3.5 Dependencies Among Commands

In simple environments, commands are usually independent of each other. These environments typically do not allow the user to modify created constructs and also it is not allowed to derive one construct from another or build up relations among various constructs.

Example 3. *For instance, we can consider an editor of plain text document. It has only two commands - INSERT, which inserts a letter to the current caret position and DELETE, which deletes a letter on current caret position. The command itself should store only the global position of the operation, which can be done using dynamic pointers. [9] (This technique takes into account shifts of text in document.) In that case, all commands are independent of each other and potentially they can be undone in any order.*

Of course, this is the ideal situation for a developer, but in real-word applications environments are much more complicated. Consider the mentioned text editor, only a bit extended - it supports simple text formating like various fonts and text size. A new command called MODIFY has to be introduced, which takes a position, sets a new format and stores the old format for undo. In this situation, the statement "..., all commands are independent on each other and potentially they can be undone in any order." is not valid anymore. If a letter is inserted and then its format is modified, the undo of INSERT should not be done before undo of MODIFY - undo of MODIFY would then changed a non-existing construct. We can see, that a possibility of modification of constructs has introduced a dependency between two commands.

Generally, there are two types of dependencies:

- *Implicit dependency*
  - If command A modifies construct C1 and older command B also modifies C1, the result of command B is dependent on the result of command A. We say, that command B depends on command A.
  - If command B depends on command A and command C depends on B, than command C depends also on A - transitivity.
- *Explicit dependency*
  - This dependency is set by the system. It complements implicit dependencies in cases, when one command uses the result of another command, but they do not work with the same construct. The transitivity also holds.

Implicit dependency does not have to be stored in a data collection, they can be computed on the fly. For this purpose it is needed to extend commands by adding information about affected construct(s). Information about explicit dependencies has to be held in dedicated data structure.

It is not important for undo/redo purposes, whether two commands are dependent explicitly or implicitly - the meaning is the same. Because of that, we just say that command A *depends* on command B.

3.6 Undo Model

Undo model represent the way the application approaches to undo and redo functionality. It describes what can be undone, how the history is logged and how the system will react to the undo operation. In some cases it may imply the way how the undo stack is visualized to the user.

In later sections of this chapter, we can observe, that particular models can differ from each other in many ways, but usually they have several common parts:

1. **Commands** - Represent solitaire actions of the user.
2. **History buffer(s)** - Store executed commands.
3. **Undo/Redo manager** - Controls history buffers.
4. **User interface** - Interacts with the user.

These parts can be easily mapped to popular architecture pattern Model-View-Controller [16]. Commands together with history buffers are the model, the user interface is the view and the controller is represented by the undo/redo manager.

All parts of the model are usually tightly coupled but the essential part of the model is the undo/redo manager - it manipulates with stacks and reacts on user's actions by selecting commands for execution and undo.

3.6.1 State of the System

State of the system (or just state) is a set of all objects in the document, including all commands, constructs and their properties.

Two states are equal if and only if all values of all objects in the document are equal.

3.6.2 Properties of Undo Models

Before we start discussion on various examples of undo models, we have to define several properties.

Stable Execution Property

Paper [6] defines a so-called *stable execution property*:

A command is always redone in the same state that it was originally executed in and is always undone in the state that was reached after the original execution. A state in this context is an ordered list of commands that are done.

This property ensures, that if a command could be originally executed and undone it would be possible also in any other time.

Weakened Stable Execution Property

The stable execution property is very strict and it holds true only for a very limited number of undo models. For this purpose, we define a *weakened stable execution property*:

During redo operation of command C, all commands, on which command C depends, are redone prior to C and during undo operation of command C, all commands dependent on C are undone prior to C.

This property also ensures, that if a command could be originally executed and undone, it would be possible also in any other time, but it is not so strict.

Stable Result Property

Stable execution property assures, that all executed commands can be undone and undone commands redone. *Stable result property* specifies, what must be the results of these operations:

- *After successful performance of undo operation on the model, there is no command in the history stack which satisfies both conditions:*
  - *It can be undone.*
  - *Calling undo on this command would cause performing an operation on non-existing object.*
- *After successful performance of redo operation on the model, there is no command in the history stack which satisfies both conditions:*
  - *It can be redone.*
  - *Calling redo on this command would cause performing an operation on non-existing object.*

If this property holds true, undo model is safe from application perspective - it should not cause a application crash because of invalid references or pointers pointing to non-existing objects.

Commutative Undo Property

The undo model is commutative if and only if the state reached after undo or redo any two commands C1 and C2 is equal to the state reached after undo or redo the same commands in the opposite order.

Commutativity ensures, that the result of any two operations in not dependent on the order, in which operations are performed. This property is important to the user, because results of models for which commutative undo property does not hold true, can be confusing.

Minimalistic Undo Property

The undo model is called minimalistic, if redo operation of command C redoes only command C and all commands older than C, on which C depends, and if

undo operation of command C undoes only the command C and all commands younger than C, which are dependent on C.

Minimalistic undo property ensures, that undo and redo operation selects the minimal possible number of commands to undo or redo. In other words, if command is not dependent or not depends on C, it will be left untouched.

4. Related Work - Current Undo Models

During past years, several undo models were presented. This chapter will focus mainly on models, which use commands as the basic unit on which undo and redo is performed. There are also other approaches, for instance saving the whole state of document after each action, but they are beyond the scope of this thesis.

The basic division of undo models is to linear undo and non-linear undo.

4.1 Linear Undo and Redo

The *linear undo model* is the simplest (and probably the most widely used) way how to achieve undo functionality. The word linear means, that only the most recently executed command can be undone.

All executed commands are stored in one history buffer. History buffer is a simple LIFO container. A new command is always pushed to the top of the buffer. This data structure is also often called *stack*.

4.1.1 Performing Undo and Redo

The user is able to undo only the most recently executed command, which is the command on the top of the stack. Such a command does not have to exist, because the stack may be empty (after document creation or load) - in that case undo is impossible and it should be signaled to the user. When undo is performed, the undone command must be prepared for redo.

The redo order follows the idea for undo operation - only the most recent undone command may be redone. Figure 4.1 shows the linear undo from user's perspective.

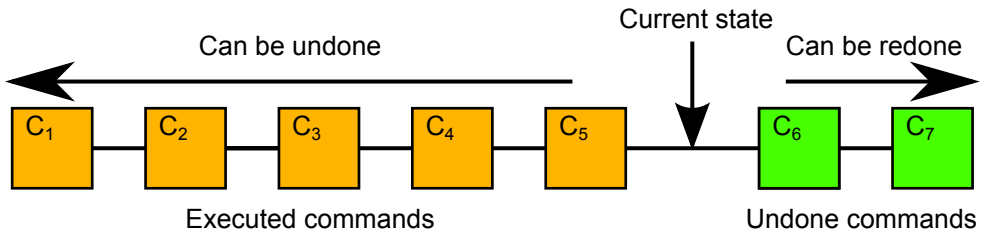


Figure 4.1: Linear undo from user perspective

4.1.2 Possible Implementation

This behavior can be achieved in two ways. Either undone command is pushed to another stack (so-called "redo stack"), or the original undo stack has pointer to the top, which is moved one command down after undo operation.

Two Stack Version

In this case, the command for redo is located on the top of the redo stack - each undone command becomes "the most recent undone command" and therefore it is pushed to the redo stack. After performing the redo, the command is returned back to the top history buffer (undo stack).

If a new command is executed and redo stack is not empty, all commands in the redo stack are discarded.

An example of this solution is depicted in Figure 4.2.

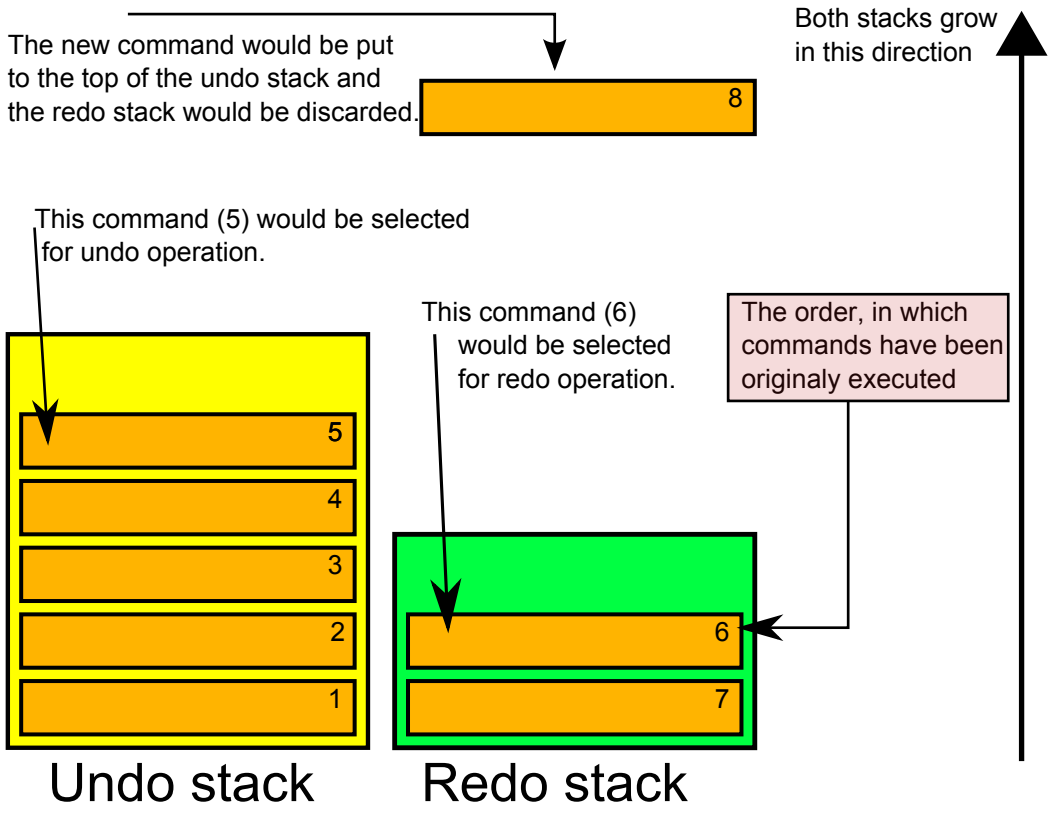


Figure 4.2: Linear undo model with two stacks

Pointer Version

The history buffer contains a variable *top pointer*, which points between the two commands - so in this case the term *top of the stack* refers to the place, where the top pointer points. The history buffer is thus split into two parts - above and under the top pointer. The part under the top pointer is full of executed commands and the part above is full of undone commands. The new command is always placed one position above the top pointer and the top pointer is then moved also one position up.

When the user wants to undo an operation, the command directly under the top pointer is taken, the undo method is called and the top pointer is moved one command down - the undone command will be above the top pointer. Redo is done in a similar way - the command right above the top pointer is taken and executed (redone) and the top pointer is moved one command up.

If a new command is executed and all commands above the top pointer are discarded the new command is placed as usual.

The example of this solution can be seen in Figure 4.3.

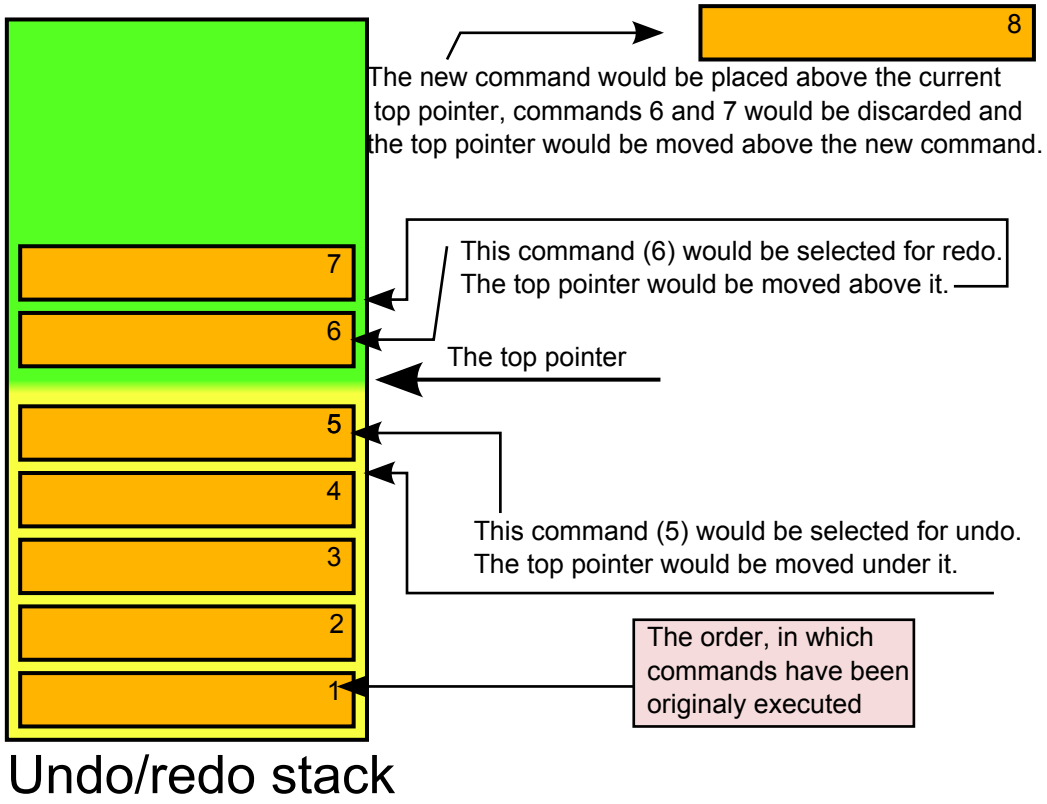


Figure 4.3: Linear undo and redo with one stack and the top pointer

Comparison of Pointer and Double Stack Version

Both versions act from user's perspective in the same way. The version with two stacks is easier to implement - there is no pointer to the top, push() and pop() methods only add or remove a command from the stack. On the other hand, there are two stacks, which lead to more memory consumption.

4.2 Non-linear Undo and Redo

A simple *linear undo* is sufficient in many applications, but it has its limitations - there is at most one *command* to undo and one to redo.

A non-linear undo brings one fundamental feature - there is a possibility of undoing or redoing also other *commands* than the last executed one. This does not imply, that any *command* can be undone at any time. Several models have been presented, each deals with the undo and redo in its own way.

4.2.1 Script Model

This model was presented by Archer et al. in 1984[5]. The core idea is simple - the history buffer is a script of commands, which is being edited by the user.

In the original version the user was really meant to edit the script as a text file, which is not very user-friendly approach (the user of a visual editor probably does not want to learn at least syntax of some script), but it would not be hard to present this script to the user in some graphical form.

All executed commands are appended to the script in ordinary way. Undo, here called *recovery*, and redo are provided by editing the script. There are three basic operations over the script:

1. *append* - Appends a new command to the script.

2. *truncate* - Removes last executed command from the script. There is also operation *truncate** which removes continuous sequence of commands from the end of the script. In fact, truncate is a special case of truncate* - truncate[1]. The removed commands may be voluntarily stored for future reuse.

3. *re-append* - Takes the initially truncated command and appends it to the script.

Editing of the script may be restricted in several ways. The restriction implies behavior and usability of the model:

- *Single-truncate* - Only append and truncate operations are permitted, and truncate cannot be used in two consecutive cycles. This restriction allows only the last executed command to be undone and redo is not supported at all.

- *repeated-truncate* - Only append and truncate* can be used. Classic approach, which mimics linear undo behavior, redo is still not supported.

- *Truncate/re-append* - All three operations are permitted. Truncated commands are stored in an auxiliary script and they can be reapended in future. The auxiliary script is not deleted after a new command is appended. This version is more powerful than classic linear undo, because it allows using a truncated command even after a new command was appended to the script. This mode is the most interesting one.

During the work with the application commands are created, appended to the script and executed. The script incrementally grows in the same way as the history buffer does - until this point, script model acts in a usual way. But when undo is initiated, the so-called *complete rerun* strategy is used.

The complete rerun means, that the state of all constructs is set to the initial one and the modified script is executed, starting with the oldest command. Another option is to perform only partial rerun - there are several checkpoints, when the state of constructs (or the whole project) is saved, and rerun is then performed only from the nearest checkpoint.

The interesting thing about this model is, that the programmer does not have to define inverse commands, which negate the impact of the original command. A command may really be represented by one dummy class, which does not store any other information. This approach can be very useful, when there are no dependencies among different commands on the stack (for example text editors

typically have no dependencies). In this situation, the behavior may be similar to the selective undo.

The biggest problem of this model is the complete rerun strategy. During the user's work, script may become very long and execution of all commands may take a serious time, especially for embedded systems. The partial rerun improves the performance in a situation, when the user wants to undo a command not so far away from the current state. The price is higher memory consumption, because more states of the project have to be saved.

4.2.2 The US&R Model

The US&R model [18] was first presented in 1984 and mainly extends the redo functionality of traditional linear undo and adds a special action to undo and redo - *skip*. Commands (called *actions* in this model) are stored not in the classic flat history buffer, but in a *history tree* - a special buffer, which allows the user to create branches in each of its nodes. There is no limit for the number of branches, nor there is a limit for the length of one branch. Each branch is again a history tree. Exactly one branch is marked as *active branch* - all executed commands will be appended to this branch.

The history tree behaves during undo similarly to the history buffer - it performs undo on commands in the active branch (let us call it A and denote the commands in this branch A1, A2...) one by one. The main difference can be seen when the user invokes a new command and there is at least one command ready for redo. Linear undo model would discard all commands to redo and the new command would replace them. The US&R model does not discard these commands, but another branch in the history tree is created (let us call the branch B, commands in this branch will be denoted B1, B2 and the command in which branch B was created AB_branch). This branch is set as "active", which means, that all other actions will be performed in this branch. If the user later decides, that the undone commands in branch A were actually correct, he/she can re-invoke them by undoing all commands from B branch back to the point AB and then calling a redo. In this situation, there are both branches available, so the application has to ask the user, which path is the right one. The user chooses A and redoes all the commands from A. An example of this behavior can be seen in Figure 4.4.

There is another feature in this model, which makes it very powerful - the *skip* operation. This operation works only with undone commands and does only one simple thing - it jumps over the command without redoing it, making the next command in branch (if any) ready for redo. Typical usage would be to undo commands to a certain point and then performing redo of "wanted" commands and skipping unwanted. In fact, this simple action allows the user to undo any command in the history tree which makes this model suitable to "selective undo ready".

4.2.3 The Triadic Model

The triadic model [19] was presented in 1988 and it is another model, which allows the user to undo any command in the history buffer.

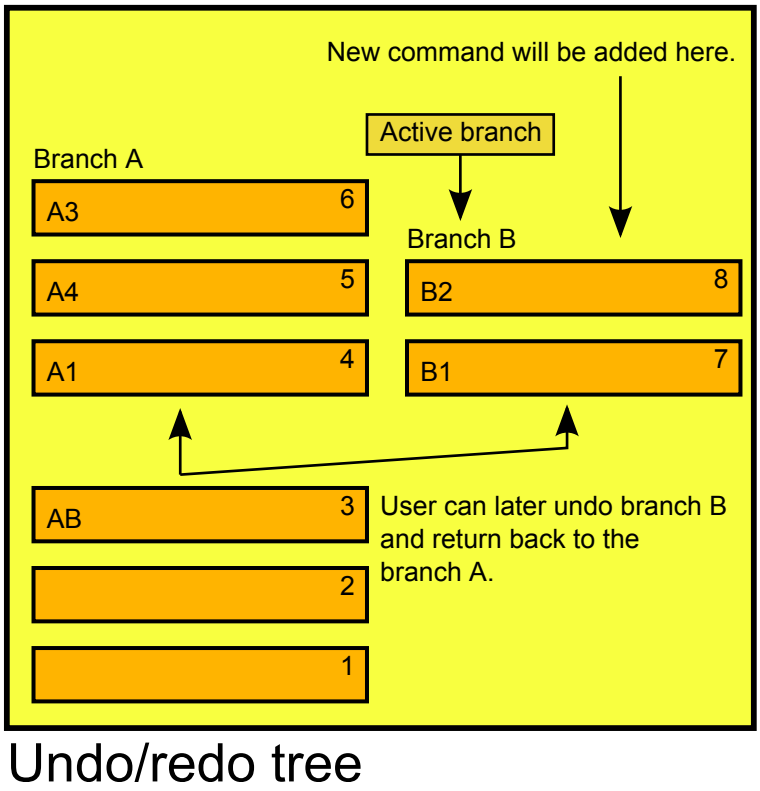


Figure 4.4: Branching with the US&R model.

The approach of triadic model is a mix between linear undo and script model. The model uses flat history buffers, which are only storage for commands. Executed commands and undone commands have separate buffers. From users' perspective, there are three visible operations:

- **Undo** - Takes the top command from *executed commands buffer*, executes the inverse operation of this command and moves the command to the top of the undone commands buffer.
- **Redo** - Takes the top command from *undone commands buffer*, executes the command and moves it to the top of the executed commands buffer.
- **Rotate** - Takes the top command from undone commands buffer and moves it to the bottom of the buffer.

These operations are originally defined more formally and support multiple arguments (such as number of commands to undo and which buffers should be affected).

The first execution of command is almost similar to the linear undo model - the command is executed and pushed to executed commands buffer. The only significant difference is, that undone command buffer remains untouched, even if it is not empty. The undo operation is performed in a similar way as in the linear undo model - only the top command from executed commands buffer can be popped (taken away) and undone. The real power of this model comes with the undone commands buffer and rotate operation. The user can undo any number of commands (which also means transfer of this command to the undone commands

buffer) and then rotate the buffer. By rotation of buffer, the user can select any command to redo. The authors claim and prove that this model allows:

- **Retraction** - Reverts the effects of the last commands.
- **Insertion** - Inserts a sequence of commands into the middle of the executed commands buffer.
- **Extraction** - Extracts a sequence of commands from the middle of the executed commands buffer.
- **Replacement** - Replaces a sequence of commands in the middle of the executed commands buffer by another sequence of the commands.
- **Transposition** - Switches positions of two sequences of commands in the executed commands buffer.
- **Reversion** - Allows the user to redo a command even if new commands were executed.

This model is interesting in the fact, that it uses inverse operations to undo commands, but the outcoming results are similar to the script model. If the user wants to undo command C in the middle of the executed commands buffer, he/she must prior undo all commands above and then redo all commands except the desired one. These commands are being redone in the state, when command C has never been executed, which is the approach introduced by the script model.

Schema of triadic model is depicted in Figure 4.5.

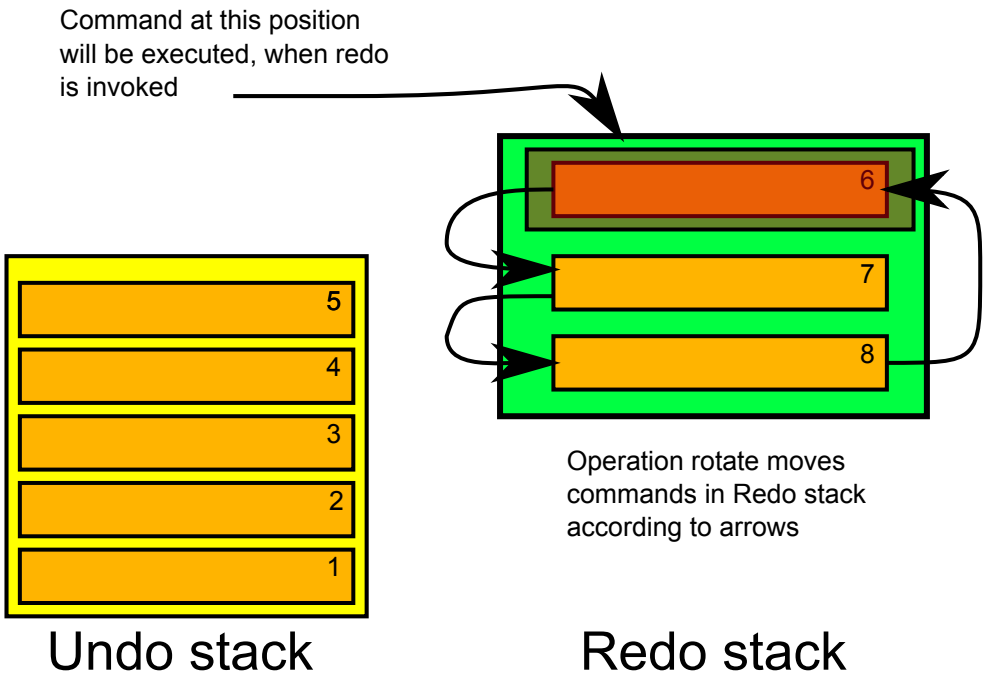


Figure 4.5: Schema of triadic models - two stacks and operations.

Model	Stable execution	Stable result	Commutative undo	Minimalistic undo
Linear undo model	yes	yes	yes	no
Script model	no	no	yes	yes
US&R model	yes	yes	yes	yes
Triadic model	yes/no[1]	yes/no[2]	no	no
Direct selective undo	no	no	yes	yes

Table 4.1: Properties of undo models.

4.2.4 Direct Selective Undo

Direct selective undo [6] is a model, which results from the classic linear undo. Execution of the commands is done in a usual way (the same as linear undo). Also the history list may be presented in the form of a simple list with the commands.

When the user picks up a command to undo (C1), this command is at first examined by the application and then a new command (C2) is created in such a way, that the effect of C2 negates the effect of C1. C2 is then executed and appended to the history buffer like an ordinary command. Redo can be performed simply - the original command is copied, executed and appended to the history buffer. This model never shortens the history buffer, even undo means appending a “”new” command to it.

The most difficult task is probably to find a command, which negates the effect of the command to undo. The technical solution can be almost similar to linear undo - the original command may have a method undo(), which returns a new command to execute. Another option is to have an undo manager, which understands the structure of the commands and therefore it can create an inverse command.

4.3 Comparison

There are six model presented, each presents one way, how the undo functionality can be done.

We start with a table 4.1 which shows properties supported by particular models.

Stable execution property can be determined very easily - if commands can be undone and redone in different document state in which they were originally executed, the property does not hold. Stable result property holds true, if the model itself cares about document stable state - if there is a mechanism which avoids unstable state of document, the property holds, otherwise not. Commutative undo property does not hold true if two identical states of history buffers may lead to two different states o the document. The minimalistic undo

[2]Yes without rotate support, no with rotate support.

Model	Undo	Redo
Linear undo model	The youngest executed.	The oldest undone.
Script model	Any executed.	Any undone.
US&R model	The youngest executed.	The oldest undone in each available branch.
Triadic model	The youngest executed.	Any undone.
Direct selective undo	Any executed.	Any undone.

Table 4.2: Which command can be selected for undo and redo

Model	Method of undo
Linear undo model	Executing inverse operation to command.
Script model	Removing a command from the history buffer, which is then replayed.
US&R model	Executing an inverse operation to command.
Triadic model	Executing an inverse operation to command.
Direct selective undo	Appending an inverse operation to history buffer and execution.

Table 4.3: Method of undo

property holds, if during undo of any command (even middle of the stack) only the requested command and commands which depend on the requested command are undone.

The next important difference between the models is which commands may be selected for undo a and which for the redo. The table summarizes 4.2 results.

The table 4.3 shows the method, which is used to negate effects of executed an command.

5. Analyses of Selective Undo

The previous chapter presented several non-linear undo models, which offer an interesting view, how undo management could be done. Some models also deal with so-called selective undo. Selective undo is not a model, but it is a feature, which a model can offer. This chapter defines this feature, discusses issues connected with the support of selective undo and offers possible solutions.

Some of the presented models have already solved these issues, some let the developers to deal with them, according to needs of the application.

5.1 Selective Undo Definition

There is no clear definition of "What selective undo is". Most papers focus on defining the correct result of the undo/redo operation and then they construct their model to give this required result. But the definition of selective undo is different, because it is not just another property, which holds true or not. The important question in this case is not "What should be the correct result of undo/redo operation?", but "What functionality should the model offer to the user?".

For the purpose of this thesis, we have selected fundamental functions the user should be allowed to do, if we want to say, that a particular model supports selective undo:

- The user should be able to undo any executed action in the history buffer. Actions independent of the action being undone should be left untouched.
- The user should be able to redo any undone action in the history buffer. Actions independent of the action being redone should be left untouched.
- No command should be ever automatically discarded from history buffer. Exception can be made on direct user's request to discard an undone command.

These requirements are common and the definition does not specify, what should be the correct result of selectively undone or redone command - there are several different ways how to achieve this functionality. Questions about the result of operations are further discussed in Section 5.2.1.

First two requirements are natural - if we talk about the **selective** undo, it should allow the user to **select** any command for requested operation. The third requirement results from non-existence of the top of the stack in case of selective undo.

5.2 Approaches and Behavior

If we look more deeply at the non-linear models presented in Chapter 4, which are suitable also for selective undo, we can see two major approaches to handling the commands in the history buffer. The first one takes history buffer as script and performs undo by returning to certain state of document and re-executing

all commands without the undone command. This approach is represented by the script model. The second approach really traverses back through the history buffer and negates or inverses the effect of undone command. The good example can be the direct selective undo.

Both approaches are able to fulfill requirements for selective undo, but results of operations can be very different.

- **Script undo models** - The correct result of undo is equal to the state of the document, in which the undone command has never been executed. The command is moved out of the script and the script is then run again without the command. The most interesting fact is that if the undone command influenced in some way any other later command, this influence exists no more.
- **Direct models** - The correct result of undo for these models is equal to the state of document, in which all commands have been normally executed and only the effect of the undone command was reversed. In other words, if there is an influence of an undone command on any other later command, this influence is still valid and it is not negated.

The basic difference between both approaches is illustrated in Figures 5.1 and 5.2, which shows a simple image editor. The initial sequence of commands is in both cases the same. The user draws circle C1, then he/she decides to fill it with red color and then the circle is copied (C2 is created) and moved left. And finally the user decides to undo command 2 "fill C1 with red color".

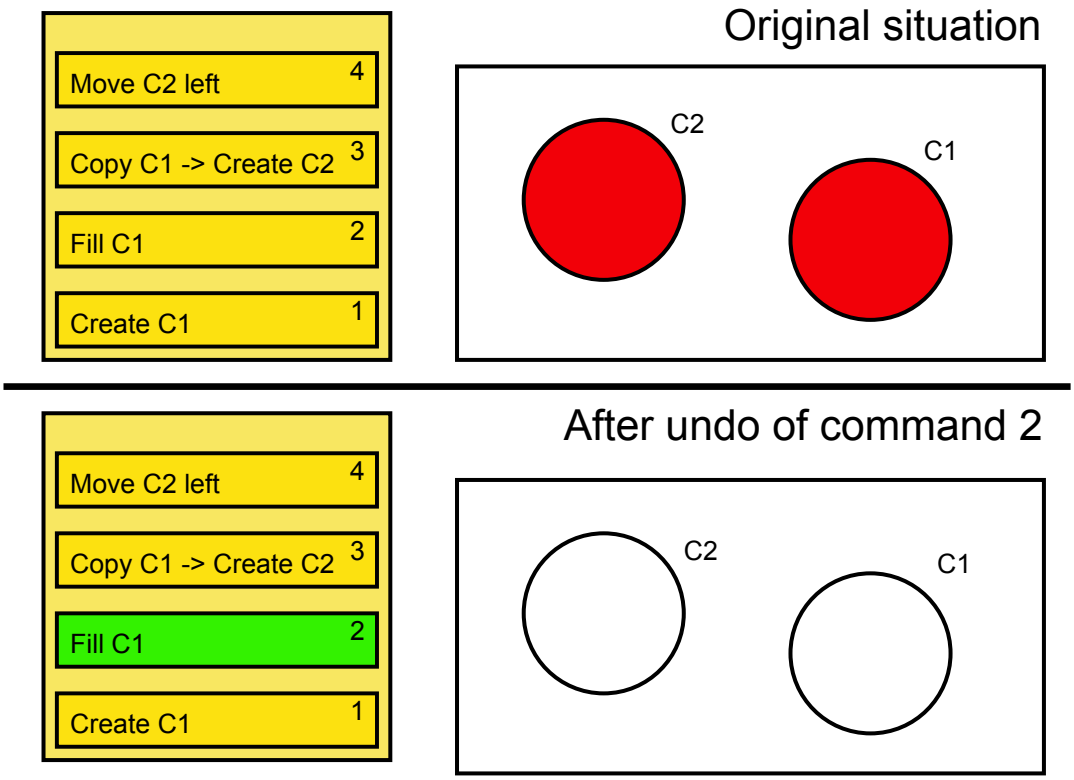


Figure 5.1: Undo with script like model.

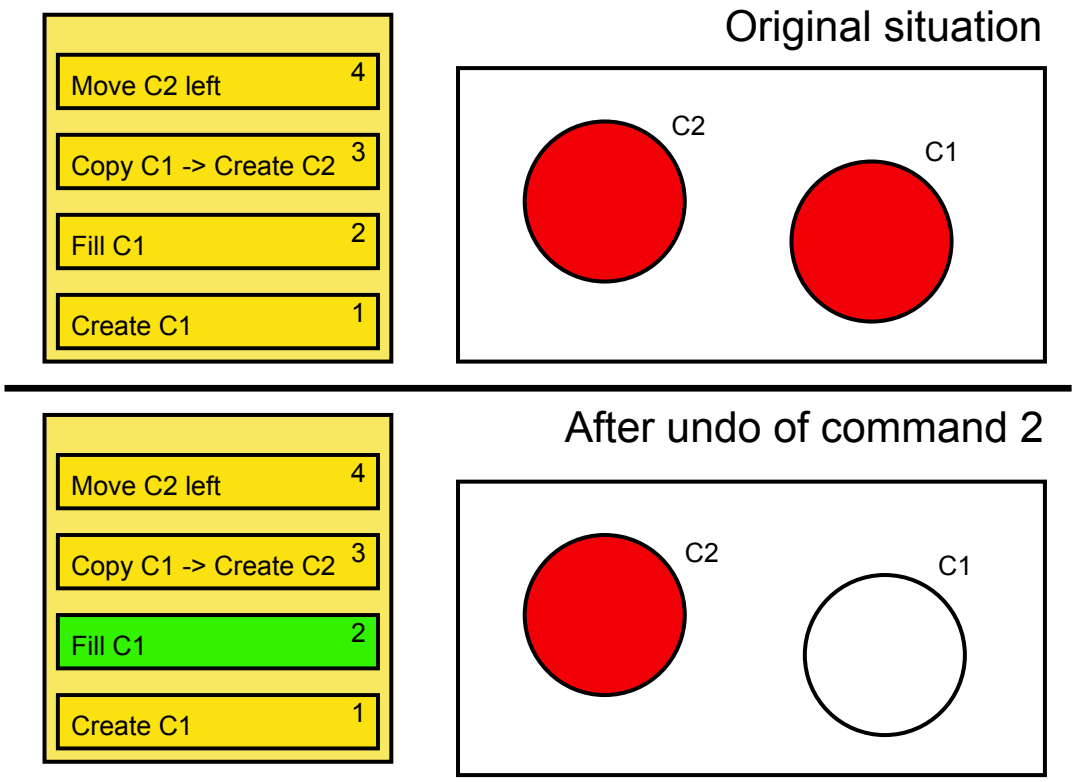


Figure 5.2: Undo with direct like model.

5.2.1 Correct and Good Result

It can be observed, that the results of undo in script-like models and direct-like models are different. Then the fundamental question is "Which one is correct?".

The answer is that both, because both models follow correctly their inner rules, how undo should be performed. The result of undo is correct as long as it brings the document into the state, in which it should be after undo operation, according to the selected model. It means, that the result of undo operation (and subsequently the state of document) is really dependent on the used undo model.

Better, but much harder, question would be "Which one is better?". This question cannot be answered until there is some definition of quality of undo result. The quality can be very individual and the approach which is appreciated by one user can be disliked by another. There is a study [8], which evaluated users' preferences for selective undo. The subjects had a figure with several shapes and the order, in which the shapes were drawn. The question was, what is the result after undoing particular steps. The majority of subjects has chosen a result similar to cascade selective undo. The problem of study is a lack of subjects (only 29 correct responds) so the results are not very predicative.

For the users, it is generally important to be able to estimate the result of the operation. The cooperation with the machine, which one cannot master, can be very frustrating and the users usually do not enjoy work with such an application [14]. For this reason we can assume for the purpose of this thesis, that good result is the one, which can be easily predicted by the user of the application. Because even if the model acts a in really strange way, it can be used as long as it acts like the user expects. When the user cannot predict the result of undo operation,

either due to error in the model itself or due to the complexity of the model, the model is unusable.

From this perspective, better results come from models similar to direct selective undo model - they are focused on inverting the effect of an single action. Results of script-like models can become very complex, especially when the user is undoing a command, which is deeply nested somewhere at the bottom of history stack.

5.3 Issues

Almost all models for selective undo need to solve several issues, which all come from one simple fact - commands may be undone or re-executed outside the context, in which they were originally executed - they break the stable execution property.

This fact is not a problem as long as there are no dependencies among commands. Regardless the chosen approach and model, as long as the commands are perfectly independent of each other, they can be undone in any order. Depending on the application, it is also possible that the commands may be shuffled and re-executed in any order.

But the real-world applications usually do not use such a simple environment. As illustrated in Chapter 3, even very simple applications like text editors with a support of text formating bring dependencies among commands and undoing commands in other order than linear may lead to unstable state of the document or program.

There are three main issues - *dead references*, *modify already modified* and *discard commands problem.*

5.3.1 Dead References

The dead references are probably the most serious problem and the next two issues are sometimes considered as sub-issues of this one.

In case of using (well-coded) linear undo, it can never happen, that some command would try to modify or destroy a construct, which does not exist. The linearity assures that every action will be undone or re-executed in the same document state in which the document was during the original execution [6]. This implies, that if the command was once executable, it will be executable also during redo.

But the selective undo functionality cannot be achieved with linear undo model. Commands may be undone in significantly different contexts than they were originally executed. The main problem is, that constructs which are being modified by these commands may be in different states or may not even exist.

Destroy or Modify Non-existing Construct

The problem is as follows:

We have a construct A, which has been created using command C1. Construct A is later modified by commands C2 and C3. Finally, the user decides that construct A is not needed and it is destroyed by command C4. Construct A no

longer exists, so every reference or pointer to it is invalid. Then, the effect of undo C1, C2, C3 is not clearly defined, because they would modify or destroy a construct, which no longer exists.

It is probably the most common issue which has to be handled.

Create Already Existing Construct

This issue is a bit artificial, but it can occur in case of successful solution of "destroy or modify a non-existing construct" issue. There is a situation in which one construct may be created twice:

The basic situation is similar to the previous one - we have a construct A, which has been created using command C1, modified by C2 and C3 and finally destroyed by C4. Now suppose, that we are able to undo C1, whose effect is silent success (it does nothing). In that case, there are two commands in the history buffer, which create construct A (redo C1 and undo C4). If both actions are performed, A is created twice.

Possible Solutions

There are two basic solutions for dead references and they differ in the level, where they are used:

- **Smart commands** - The design of the command is the fundamental part of each undo model and command structure can determine abilities of particular models. Commands tend to be minimalistic. This requirement is understandable - commands are quite common in the sense of quantity and during a long work with application, lot of (thousands of) commands may be generated. If each generated command is stored in the history buffer, the memory consumption may grow into unacceptable values, especially for embedded devices.

  The goal is to design a command in such a way, that calling the undo() or execute() method can deal with dead references. Each command at first checks, whether all references or pointers are valid, i.e. the affected objects really exist. Only if this check is successful, the requested action is performed. If some reference is marked as dead, there are again two possible solutions:

  - Silent success - The command does nothing and it is marked as if the action was performed). The command must not have any side-effects, which are not related to the modified construct, because these side-effects would be also skipped.
  - Error is raised - The user is informed, that the action cannot be performed. This solution is technically less challenging and easier to implement, but it can be annoying to the user. The error information may supply also a possible solution of this situation which helps the user to get the document into desired state, although it negates the simplicity of this solution.

- **Cascade undo** - This solution is not command centered, but the logic is implemented in the undo manager. The manager executes and undoes commands in such a way, that there will be no dead reference in the history buffer. The commands have to provide information about constructs they affect, on which commands they depend and what is the action they perform (create, modify, destroy). With these information, the manager is able to determine the correct action:
  - Undo of create construct command - All commands, which affect the construct should be also undone. The construct will no longer exist, so there should be no command to undo, which affects it.
  - Undo of modify construct command - If there is a destroy command for the modified construct, it should be undone. The construct must exist, so if it was destroyed, it must be re-created (by undoing the destroy command).

The *smart commands* solution can be used for both script-like and direct-like model. Script-like models would probably prefer the version with silent success, because it does not interrupt script run. Cascade undo is more natural for direct-like models, because their history stacks are able to search commands anyway.

5.3.2 Modify Already Modified

This issue does not affect only selective undo models and it is rather logical than technological. The situation is as follows:

We have a construct A which has property P of type string. After the creation of A, P has value "abcd". Then two commands C1 and C2 are invoked, both modifying the value of P, C1 sets it to "efgh" and C2 to "ijkl". Now consider the selective undo of C1. The question is, what is then the correct value of P?

Basically, there are two possible values - "ijkl" or "abcd". The script-like models will restore the document to some state before execution of C1 and then they will execute all commands without C1, including C2, which will set value of P to "ijkl". The effect of direct-like models depends on actual implementation of commands, but a common solution would call the undo method of C1, which would change the value to the previous one (from C1's perspective) - "abcd".

Until this point, both types of models behave in an expected way. But if we go further:

After successful undo of C1, the user calls also undo on C2.

Script-like models will not touch the original value of P, because both commands are now undone and so it remains "abcd". The direct-like models will call undo operation on C2, which similarly to C1 restores the previous value - now "efgh". This situation is interesting - both C1 and C2 are undone but value of P is equal to the state after execution of C1.

This little paradox results from the basis of the direct-like model - the result of selective undo may really be dependent of the order, in which all commands are undone (or redone). The value of construct or its property is set by last executed or undone command, which affected this construct or property. This issue is also discussed in [14].

Possible Solutions

In case of script-like models, there is nothing to solve, it behaves in an expected way. Direct-like models have two solutions:

- **Let it be.** - This behavior can be considered as strange from user's perspective, but it does not lead to unstable document or program state. The user should be informed in some way (for instance on the visualization of the history stack), what will be the result.
- **Voluntary cascade undo.** - All younger modification commands, which affect the same construct, are also undone.

5.3.3 Discard Commands Problem

In linear undo model, commands above the stack top are usually discarded from the stack after a new command has been executed. This behavior is necessary, because there must be a place for the newly executed command and executed and undone commands cannot be mixed together.

The selective undo definition specifies, that no commands should be discarded from the stack. It is necessary, because selective undo models usually do not have a pointer to the top of the history buffer and therefore the model cannot determine, which commands should be discarded.

But there is a problem - when the user undoes a command, his/her intention is to negate the effect of the command. The effect of a command is negated, because the effect is probably unwanted by the user and if effect is unwanted, the command will probably not be redone again - redo is often used just to correct badly selected undo operation. If this logic is true, the history stacks can be filled with many undone commands, which will never be redone, because their effects are useless for the user. This fact will make orientation in the stack harder and the user may spend valuable time just by searching, which command should be undone.

Possible Solutions

- **Discard operation** - The user can manually select a command, which will be discarded from the history buffer.
- **Linear-like discarding** - If there is a continuous sequence of undone commands at the end of the buffer, all commands in this sequence will be discarded when a new command is executed.
- **The oldest must go** - Each history buffer holds only limited number of commands and when this number is reached, the oldest command in the buffer (regardless its state) is discarded.

It should be noted, that this issue does not have to be solved, because it will not bring the document into an unstable state. However, possibility of discarding commands from the buffer can simplify work with the model.

6. Proposed Algorithms

This chapter presents three models for undo in complex environments. It starts with more precise definition of the environment and then defines the models.

6.1 Environment

Before we can start with algorithms, we need to clearly specify the environment, in which our models will operate. This environment aims to match real-world applications as much as possible.

6.1.1 Document

The most high-level unit of environment is a *document*, also called *project*. It can be considered as a giant container, which creates a big envelope around all other objects. It voluntarily provides methods to serialize and de-serialize its content (usually used to save and later load the work), print results, convert the content to some interchange format.

A document is a standalone unit - if a system supports simultaneous work with multiple documents, they are independent and cannot interfere with each other.

It is also common, but not mandatory, that documents can be transfered to other instances of the application.

Example

The good examples of documents are:

- Text file - A set of characters.
- UML project - A set of UML classes and relations among them.
- Scalable vector graphics [2] file - A set of defined graphical objects and shapes.

6.1.2 Constructs and Properties

Constructs are defined in Section 3.1 and they are used according to this definition.

All constructs reside in document's construct pool, which is some suitable data structure used to store constructs. Constructs are shared among workspaces and one construct may be visualized on several workspaces.

6.1.3 Workspace and History Buffers

A workspace is an interface for the user, which he/she can use to modify the content of the document. One document may be modified from several workspaces and also one construct may be viewed in several workspaces.

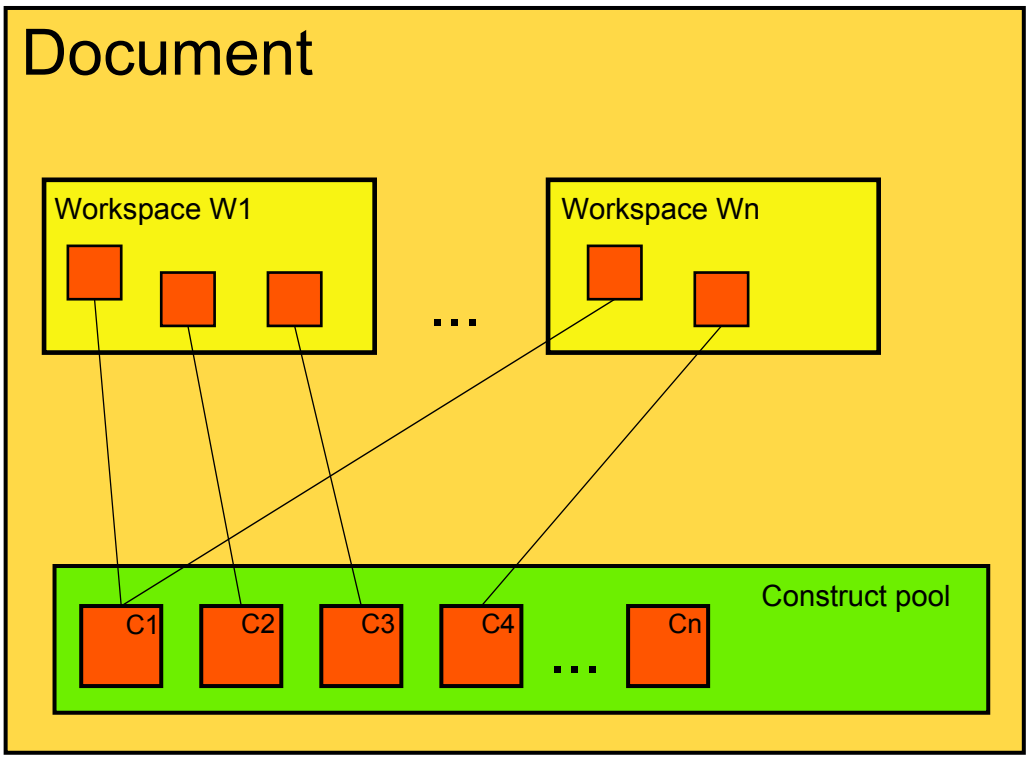


Figure 6.1: Schema of one document with several constructs (C1..Cn) and workspaces(W1..Wn).

6.2 Extended Linear Undo

The *extended linear undo* model is the first new undo model presented in this thesis. The main motivation for creating this model was a need to have model, which is able to manage undo and redo in a safe and user-friendly way in case of several connected history buffers.

6.2.1 Requirements

This model is the most basic one. It should be intuitive for the user - act like the usual linear undo model does - and easy to implement in existing applications. If an undo will be done in one workspace, it should not affect other workspaces, unless it is necessary.

The properties to hold true should be:

- **Weakened stable execution property** - Although selective undo is not the goal, stable execution property would mean implementing global linear undo over all history buffers, which is not the aim.
- **Stable result property** - Model would not be usable without satisfying this property.
- **Commutative undo property** - Important for intuitive usage.

The behavior of the algorithm results from linear undo model:

1. At most one command will be offered in each workspace for undo and redo.
2. Command offered for undo will be the oldest executed command in the history buffer.
3. Command offered for redo will be the youngest undone command in the history buffer.
4. After successful call of execute operation on command C in workspace W, command C will be executed and it will be the first command to undo in workspace W.
5. After successful call of undo operation on command C in workspace W, command C will be undone and it will be the first command offered to redo in workspace W.
6. After successful call of redo operation on command C in workspace W, command C will be redone and it will be the first command offered to undo in workspace W.
7. All commands to redo in workspace W are discarded after a successful execution of a new command in workspace W.

If the model satisfies all the properties and acts according to the specified behavior, we can say, that it is *correct*.

6.2.2 Principle and Analysis

The key idea of the extended linear undo is to take a plain linear undo model and modify it in the way, that it suits for environment with several workspaces.

To satisfy intuitiveness and easy implementation for existing applications, it would be useful, if each workspace would manage its own history buffer - if a command is created in the workspace, it goes to the top of its buffer. This approach results from one assumption: *If a user is working in particular workspace, he/she wants to undo or redo commands executed in this workspace prior to commands in other workspaces.* It obeys locality of users' work and usually the user does not switch workspaces very often so it will bring the user closer to feel like he/she is using regular linear undo model - he/she can undo or redo only the commands, which are right bellow or above the stack top.

But we cannot simply perform undo in one stack without checking other stacks for possible dependent commands - this would break the stable result property. For this purpose, we need to know exact order of commands among stacks - it can be simply achieved by sequence numbering of each command. During undo of command C, we have to check all younger commands, if they depend or not on C. This check must be done recursively, because all dependent commands will have to be undone and also their dependencies need to be checked. A similar situation is for the redo operation.

If there is at least one dependency to an other stack found, an action needs to be performed:

- In case of **undo** - Let us call the first dependent command in the particular history buffer C_u. Then all commands in the history buffer above the C_u need to be also undone.
- In case of **redo** - Let us call the first dependent command in the particular history buffer C_r. Then all commands in the history buffer under the C_r need to be also redone.

This behavior will be very similar to the linear undo, because in each history buffer there will be just one continuous sequence of executed commands and one continuous sequence of undone commands.

6.2.3 Data Structures and Algorithm[1]

At first we have to specify the data structures, that will be used in the algorithm.

Command

```
struct {
        int sequence_number;
        int keys_of_affected_constructs[];
        bool is_undone;

        void* user_data;
} Command;
```

We use the extended variant of command. `sequence_number` is a global unique numbering among all commands, each created command has this increased by one in respect to its predecessor. The array `keys_of_affected_constructs` is a collection of all constructs' keys, which are either created, modified or destroyed by this command. `user_data` is a pointer to the implementation-specific data structure used to perform actual work of command.

There are three functions directly related to `Command` structure:

- `Execute(Command)`- Executes command for the first time (Algorithm 1).
- `Undo(Command)` - Undoes the effect of executed command(Algorithm 2).
- `Redo(Command)` - Is used for subsequent executions, may be the same as `Execute` (Algorithm 3).

Implementation of these three function are simple, see Algorithms 1, 2, 3.

Algorithm 1 Execution of command

```
1: procedure EXECUTE(command)
2:     command.is_undone ← false
3:     DoExecute(command.user_data)
4: end procedure
```

[1]Data structures are written in C-like syntax, procedures in pseudocode.

Algorithm 2 Undo of command

```
1: procedure UNDO(command)
2:     command.is_undone ← true
3:     DoUndo(command.user_data)
4: end procedure
```

Algorithm 3 Redo of command

```
1: procedure REDO(command)
2:     command.is_undone ← false
3:     DoRedo(command.user_data)
4: end procedure
```

Functions `DoExecute`, `DoUndo` and `DoRedo` are command specific and their implementation is dependent on command's purpose. They perform the requested actions. In a programming language with a support of classes and virtual inheritance, these function (methods) would probably be virtual and overridden for each particular class of command.

History_buffer

```
struct {
        Command command_stack[];
        int index_of_top;
        int key_of_workspace;
} History_buffer;
```

`History_buffer` is also a simple structure. It consists of collection of commands `command_stack[]`. `Index_of_top` specifies the position, where top of the stack is - boundary line between executed and undone commands. It it the position in an array, where a new command can be added, in case of empty `command_stack` the value is 0. When a new command is executed and pushed to the stack, it is increased by one; when a command is redone, it is also increased by one; when command is undone, it is decreased by one. `key_of_workspace` serves as a unique identifier of buffer.

There are also several functions, which control the history buffer. For simplification, if a function argument is of type `General_collection`, it can be called also with type `history_buffer` and then it operates of `history_buffer.command_stack`.

- `ItemCount(General_collection)` - Returns the number of items in general collection.
- `Pop(General_collection)` - Removes the last item of collection.
- `Push(General_collection, Item)` - Adds one item to the end of the collection.
- `GetCommandToUndo(workspace_key)` - Returns the first command under the top of history stack (see Algorithm 4).
- `GetCommandToRedo(workspace_key)` - Returns the first command on which the top of the history stack points(see Algorithm 5).

- `DiscardRedoCommands(workspace_key)` - Removes all command above current top oh the history stack (see Algorithm 6).
- `MoveTopPointerUp(workspace_key,n)` - Moves pointer to the top of the history buffer n positions up.
- `MoveTopPointerDown(workspace_key,n)` - Moves pointer to the top of the history buffer n positions down.

Functions `ItemCount`, `Pop` and `Push` are usually provided by the programming environment or external library for managing collections, for implementation of other function see Algorithms 4, 5, 6.

Algorithm 4 Get command to undo

```
1: procedure GetCommandToUndo(workspace_key)
2:     hist_buff ← GetHistoryBuffer(workspace_key)
3:     if hist_buff.index_of_top = 0 then
4:         return NULL
5:     end if
6:     return hist_buff.command_stack[hist_buff.index_of_top - 1]
7: end procedure
```

Algorithm 5 Get command to redo

```
1: procedure GetCommandToRedo(workspace_key)
2:     hist_buff ← GetHistoryBuffer(workspace_key)
3:     if hist_buff.index_of_top = ItemCount(hist_buff.command_stack) then
4:         return NULL
5:     end if
6:     return hist_buff.command_stack[hist_buff.index_of_top]
7: end procedure
```

Both functions `GetCommandToUndo` and `GetCommandToRedo` at first check, whether there is a suitable command to undo or redo and then returns NULL or the desired command. `index_of_top` is always index to the position, where a newly executed command should be placed.

Algorithm 6 Discard all commands to redo in workspace

```
1: procedure DiscardCommandsToRedo(workspace_key)
2:     hist_buff ← GetHistoryBuffer(workspace_key)
3:     for i ← hist_buff.index_of_top to ItemCount(hist_buff.command_stack) do
4:         RemoveIndexFromCollection(hist_buff.command_stack, i)
5:     end for
6: end procedure
```

This function simply takes all commands above the `index_of_top` and removes them from the collection `command_stack`. `MovePointerUp/Down` only increases/decreases the value of variable `index_of_top` and therefore the implementation would be simple.

Global History Buffer

As stated in Section 6.2.2, it will be needed to search through all commands in the system for possible dependencies. For this purpose, we create a collection of pointers to commands.

```
struct Command* global_history_buffer[];
```

This collection is ordered in an ascending manner by `sequence_number` of commands and serves for simpler searching for dependencies - the function can just iterate through the collection. In real implementation it can be replaced by a linked list in the structure of command or just dropped, but searching function will then have to find firstly on which stack is the command with the lowest following `sequence_number`.

6.2.4 Algorithm

The structures presented in the previous section are the essential ones, which are needed to describe the algorithm. We start with two helper functions, which iterate through `global_command_buffer` and search for dependencies - Algorithms 7 and 8.

Algorithm 7 Undo dependencies searcher for Extended linear undo model

```
1:  procedure FindDependentCommandsForUndo(command_to_undo)
2:      index ← GetIndexToGlobalBuffer(command.sequence_number)
3:                                                  ▷ Affected construct keys
4:      ack ← command.keys_of_affected_constructs
5:                                                  ▷ Affected history buffers
6:      ahb ← GetHistoryBufferKey(command)
7:                                                      ▷ Commands to undo
8:      commands2undo ← command
9:      for i = index + 1 → ItemCount(global_command_buffer) do
10:         comm ← global_command_buffer[i]
11:         if comm.is_undone then
12:             continue
13:         end if
14:         if ahb ∩ GetHistoryBufferKey(comm) then
15:             ack ← ack ∪ comm.keys_of_affected_constructs)
16:             commands2undo ∪ com2undo ∪ command
17:         else if ack ∩ comm.keys_of_affected_constructs then
18:             ack ← ack, ∪ comm.keys_of_affected_constructs)
19:             ahb ← ahb, ∪ GetHistoryBufferKey(comm))
20:             commands2undo ← commands2undo ∪ command
21:         end if
22:     end for
        return commands2undo
23: end procedure
```

This procedure takes only one argument - the starting command. It incrementally builds up three collections:

1. Affected construct keys (`ack`) - Initialized with the `keys_of_affected_constructs` collection of the starting command.

2. Affected history buffers (`ahb`) - Initialized with the key of history buffer of the starting command.

3. Commands to undo (`commands2undo`) - Initialized with the starting command.

It iterates in ascending order through all the commands in the system and searches for commands which:

- are located in the buffer, which is in affected history buffers. (If so, it means that at least one construct from this buffer was previously selected for undo.)

- affect one of constructs which are in affected construct keys. (If so, it means that the command is dependent on some other command, which was previously selected for undo.)

If command C_{toUndo} satisfies at least one of these conditions, it is added to the `commands2undo` collection, constructs C_{toUndo} modifies/creates/deletes are merged with `ack` collection and the buffer, in which C_{toUndo} resides, is added into `ahb` collection. After iterating through all younger commands, the `commands2undo` collection contains all commands to perform successful undo.

The problem is similar in case of redo operation of command C_{toRedo}, but task is a bit different - we have to find all commands, on which the C_{toRedo} is dependent. The function `FindDependentCommandsForRedo` (Algorithm 8) does almost similar work as `FindDependentCommandsForUndo` (Algorithm 7), but backwards.

`FindDependentCommandsForRedo` returns all commands, on which C_{toRedo} command is dependent.

Once we are able to construct a collection of all commands, which have to be undone or redone, we can perform the actual execution, undo and redo. All three procedures (see Algorithms 9, 10 and 11) take as argument key of the workspace, where undo is being performed, procedure `ExecuteCommand` must also have access to particular command.

`ExecuteCommand` (Algorithm 9) is a simple procedure - it only calls function `Execute()` on its argument $C_{toExecute}$, checks whether this execution has been successful and if yes, it removes all commands above the stack top (undone commands) from the buffer, pushes $C_{toExecute}$ to the buffer and moves pointer to top of the stack one position up. After calling this function, there are no commands for redo and pointer to the top points one position above the last command in the buffer, which is $C_{toExecute}$. $C_{toExecute}$ is then appended to the `global_history_buffer` (it can be appended to the end, because it surely has bigger sequence number than any other command).

The `UndoCommand` function (Algorithm 10) function uses previously developed function `FindDependentCommandsForUndo`, which builds up a collection of all dependent commands. When a collection is built, rest of the job is simple. It is just needed to reverse the collection - the oldest command is in the first place and undo has to be done starting with the youngest command.

Algorithm 8 Redo dependencies searcher for Extended linear undo model

```
1: procedure FindDependentCommandsForRedo(command_to_redo)
2:     i ← GetIndexToGlobalBuffer(command.sequence_number)
3:                                                  ▷ Affected construct keys
4:     ack ← command.keys_of_affected_constructs
5:                                                  ▷ Affected history buffers
6:     ahb ← GetHistoryBufferKey(command)
7:                                                  ▷ Commands to redo
8:     commands2redo ← command
9:             ▷ Gets a key of history buffer in which the command is present
10:    for i = i - 1 → 0  do
11:        comm ← global_command_buffer[i]
12:        if ¬ comm.is_undone then
13:            Continue
14:        end if
15:        if  ahb ∩ GetHistoryBufferKey(comm) then
16:            ack ← ack ∪ comm.keys_of_affected_constructs)
17:            commands2redo ← commands2redo ∪ command
18:        else if  ack ∩ comm.keys_of_affected_constructs then
19:            ack ← ack ∪ comm.keys_of_affected_constructs)
20:            ahb ← ahb ∪ GetHistoryBufferKey(comm))
21:            commands2redo ← commands2redo ∪ command
22:        end if
23:    end for
        return commands2redo
24: end procedure
```

Algorithm 9 Execute command

```
1: procedure ExecuteCommand(workspace_key, command)
2:     history_buffer ← GetHistoryBuffer(workspace_key)
3:     if E thenxecute(command)
4:         DiscardRedoCommands(history_buffer)
5:         Push(history_buffer, command)
6:         MoveTopPointerUp(history_buffer,1)
7:         Append(global_history_buffer, command)
8:         return true
9:     end if
10:    return false
11: end procedure
```

The procedure `RedoCommand` (Algorithm 11) is almost similar to `UndoCommand`, but `FindDependentCommandsForRedo` is used to built up the collection of dependent commands.

Algorithm 10 Undo command

```
1: procedure UndoCommand(workspace_key)
2:     workspace ← global_workspace_collection[workspace_key]
3:     com2undo ← GetCommandToUndo(workspace)
4:                                                  ▷ Command to undo
5:     if com2undo ≡ NULL then return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo)
8:                                          ▷ Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:        history_buffer ← GetHistoryBuffer(com2undo_collection[i])
13:        MoveTopPointerDown(history_buffer,1)
14:    end for
15:    return true
16: end procedure
```

Algorithm 11 Redo command

```
1: procedure RedoCommand(workspace_key)
2:     workspace ← global_workspace_collection[workspace_key]
3:     com2redo ← GetCommandToRedo(workspace)
4:                                                  ▷ Command to redo
5:     if com2redo ≡ NULL then return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo)
8:                                          ▷ Collection of commands to redo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Undo(com2redo_collection[i])
12:        history_buffer ← GetHistoryBuffer(com2redo_collection[i])
13:        MoveTopPointerUp(history_buffer,1)
14:    end for
15:    return true
16: end procedure
```

6.2.5 Correctness

This section presents proofs, that the algorithm itself is correct and gives the required results. To prove this, we need to prove that all properties declared to hold true really hold true and the behavior matches the behavior specified in requirements section.

Functions dealing with command structure

All functions dealing directly with commands - `Execute`, `Undo` and `Redo` are considered to be correct. They only check, whether operation can be performed and then call the command-specific function.

Functions dealing with history buffer structure

The same approach can be used for most of functions manipulating with history buffer. `ItemCount`, `Pop`, `Push` are provided by implementation of the underlaying collection. `MoveTopPointerUp`/`MoveTopPointerDown` only increases/decreases a variable, which is obviously correct.

Theorem 1. *`GetCommandToUndo` returns a command at position `index_of_top`− 1 or NULL, if such a command does not exist.*

Proof of Theorem 1. To prove this, we have to evaluate function flow. At first it gets correct `history_buffer`, then it tests `index_of_top` if it is 0. If so, it returns NULL, if not, it returns `hist_buff.command_stack[hist_buff.index_of_top - 1]`, which is the command at position `index_of_top` − 1.

The `index_of_top` is initialized with zero, increased by one when a command is added and executed or redone and decreased by one, when a command is undone. Executed command at position `index_of_top` − 1 does not exist if and only if the stack is empty (`index_of_top` = 0) or all commands are undone (`index_of_top` = 0).

□

Theorem 2. *`GetCommandToRedo` returns a command at position `index_of_top` or NULL, if such a command does not exist.*

Proof of Theorem 2. `GetCommandToRedo` gets at first correct `history_buffer`, then it tests `index_of_top` if is equal to `ItemCount(command_stack)`. If so, it returns NULL, if not, it returns `hist_buff.command_stack[hist_buff.index_of_top`, which is command at position `index_of_top`.

The undone command at position `index_of_top` does not exist if and only if `history_stack` is empty or all commands in the `history_stack` are executed. If stack is empty, ItemCount is 0 and `index_of_top` also. If all commands are executed, `index_of_top` is equal to ItemCount. □

Function `DiscardCommandsToRedo` only removes all commands at higher position than `index_of_top`. It performs this operation using function provided by collection management, which is considered to be correct.

Algorithm

When we know, that functions dealing with the data structures are correct, than we can prove correctness of the algorithm.

We start with one simple invariant.

Invariant 1. *The value of `index_of_top` variable for each `history_buffer` after finishing undo, redo or execute operation is equal to index of last executed command plus one. All commands at positions less than `index_of_top` are executed, all other commands are undone.*

Proof of Invariant 1. After buffer initialization, buffer is empty, index of last executed command is not-defined and value of `index_of_top` is set to zero.

Execution of command adds one executed command to the position of `index_of_top` and increases `index_of_top` by one. Discarding undone commands at

positions higher than `index_of_top` does not affect the value of `index_of_top`. Invariant holds.

Redo of command executes command at the position `index_of_top` and increases `index_of_top` by one. Invariant holds.

Undo of command undoes the command at the position `index_of _top` -1 and decreases the `index_of_top` by one. Invariant holds.

There are no other functions which would change the value of `index_of_top` nor add or remove commands from the `history_buffer`.

□

This invariant assures, that operations performed on the one buffer do not break conditions for linear undo model.

Now, we should prove that both functions which search for dependent commands always find all dependent commands.

Theorem 3. *Function* `FindDependentCommandsForUndo` *will return a collection of commands* $Coms_{toUndo}$*, which contains all executed commands younger than its argument* C_{toUndo} *which are dependent on* C_{toUndo}*. If dependent command* C_{dep} *is found in history buffer* HB_i*, the returned* $Coms_{toUndo}$ *will contain also all commands in* HB_i *older than* C_{dep} *and younger than* C_{toUndo}*.*

Proof of Theorem 3. Suppose, that there is executed command $C_{dependent}$ younger then C_{toUndo}, which is transitively dependent on C_{toUndo} or there is an executed command $C_{sameStack}$ also younger then C_{toUndo}, which is located in the `history_buffer` HB_x, where already at least one command younger than $C_{sameStack}$ is selected to undo, and none of $C_{dependent}$ and $C_{sameStack}$ is present in $Coms_{toUndo}$ after successful call of `FindDependentCommandsForUndo`.

There are total n commands in the `global_history _buffer` which are randomly distributed into m `history_buffers`. According to Invariant 1, if there is undone command C_{undone} belonging to history buffer H_a, all younger commands belonging to H_a are also undone.

The function creates two collections - `ack` (affected construct keys) and `ahb` (affected history buffers). The first one is initialized with `keys_of_affected _constructs` of C_{toUndo} and the second one by key of `history_buffer` (same as workspace key) to which C_{toUndo} belongs.

The function iterates through all commands in `global_history_buffer` from older to younger, starting with direct successor of C_{toUndo}. Each such a command C_i is

- added into collection $Coms_{toUndo}$ AND
- its `keys_of_affected_constructs` collection is merged with `ack` collection AND
- the key of history buffer to which it belongs is merged with `ahb` collection

if at least one of these two conditions hold true:

- The set of `keys_of_affected_constructs` of C_i intersects with `ack`.
- The key of `history_buffer` to which C_i belongs intersects with `ahb`.

The first condition is true if and only if C_i affects a construct, which was already touched by one of constructs in $Coms_{toUndo}$. Because $Coms_{toUndo}$ was originally initialized with the `keys_of_affected_constructs` of C_{toUndo}, C_i is transitively dependent on C_{toUndo}.

The second condition means, that if the key of history buffer H_i, to which C_i belongs, is already in `ahb` collection, a command older than C_i from the H_i has already been selected selected for undo.

But that means, that both commands $C_{dependent}$ and $C_{sameStack}$ cannot exist, because they would be selected to undo.

□

Theorem 4. *Function* ***`FindDependentCommandsForRedo`*** *will return a collection of commands $Coms_{toRedo}$, which contains undone commands older than its argument C_{toRedo} on which C_{toRedo} depends. If a dependent command C_{dep} is found in the* ***`history_buffer`*** *HB_i, the returned $Coms_{toRedo}$ will contain also all commands in HB_i younger than C_{dep} and older than C_{toRedo}.*

Proof of theorem 4. Suppose, that there is an undone command $C_{dependent}$ older then C_{toRedo}, on which C_{toRedo} (transitively) depends or there is an undone command $C_{sameStack}$ also older then C_{toRedo}, which is located in history_buffer, where already at least one command younger than $C_{sameStack}$ is selected for undo, and none of $C_{dependent}$ and $C_{sameStack}$ is present in $Coms_{toRedo}$ after successful call of `FindDependentCommandsForRedo`.

The function creates two collections - `ack` (affected construct keys) and `ahb` (affected history buffers). The first one is initialized with `keys_of_affected_constructs` of C_{toRedo} and the second one with the key of the history_buffer to which C_{toRedo} belongs (same as the workspace key).

The function iterates through commands in `global_history_buffer` in direction from younger to older, starting with direct predecessor of C_{toRedo}. Each such a command (let us call it C_i) is

- added into collection $Coms_{toRedo}$ AND
- its `keys_of_affected_constructs` collection is merged with `ack` collection AND
- key of history buffer to which it belongs is merged with `ahb` collection

if at least one of these two conditions hold true:

- The set of `keys_of_affected_constructs` of C_i intersects with `ack`.
- The key of `history_buffer` to which C_i belongs intersects with `ahb`.

The first condition is true if and only if C_i affects a construct, which was already touched by one of the constructs in $Coms_{toRedo}$. Because the $Coms_{toRedo}$ was originally initialized with the `keys_of_affected_constructs` of C_{toRedo} the C_{toRedo} is transitively dependent on C_i.

The second condition means, that if the key of history buffer H_i, to which C_i belongs, is already in `ahb` collection, a command older than C_i from the H_i has already been selected selected for redo.

But that means, that both commands $C_{dependent}$ and $C_{sameStack}$ cannot exist, because they would be selected to redo.

□

Behavior

There are 7 points defined in requirement section, which specifies required behavior.Each one will be proved to be satisfied:

1. Both functions `GetCommandToUndo` and `GetCommandToRedo` returns at most 1 command.

2. It has been already proved in proof of correctness of `GetCommandToUndo` function, that `GetCommandToUndo` returns the youngest executed command in the `history_stack`.

3. It has been already proved in proof of correctness of `GetCommandToRedo` function, that `GetCommandToRedo` returns the oldest undone command in the `history_stack`.

4. Function `ExecuteCommand` executes a command $C_{toExecute}$, discards all commands at positions greater or equal to `index_of_top`, places $C_{toExecute}$ at the position specified by `index_of_top` and increments `index_of_top` by 1. $C_{toExecute}$ is then at the position `index_of_top` $-$ 1 and it is also the youngest executed commands in this stack $\Rightarrow$ $C_{toExecute}$ will be offered as a first command to undo.

5. Function `UndoCommand` undoes a command C_{toUndo} at the position `index_of_top`$-$ 1 and decrements `index_of_top` by 1. C_{toUndo} is then on the position `index_of_top` and it will be offered as a first command to redo.

6. Function `RedoCommand` redoes a command at position `index_of_top` and increments `index_of_top` by 1. Redone command is then on the position `index_of_top` $-$ 1 and it will be offered as a first command to undo.

7. Function `ExecuteCommand` executes a command $C_{toExecute}$, discards all commands at positions greater or equal to `index_of_top`. All commands at these positions are undone and no other undone command cannot be in stack.

Properties

Now we can prove, that the extended linear undo model satisfies the requested properties.

Weakened stable execution property. Proofs of Theorems 3 4 prove, that in case of undo operation of command C_{toUndo}, all younger dependent commands are also chosen to undo prior to C_{toUndo} and in case of redo of C_{toRedo}, all older commands on which C_{toRedo} depends are chosen to redo prior to C_{toRedo}.

Commands are always undone or redone in the right order. In case of undo, function `FindDependentCommandsForUndo` will return collection ordered in ascending order according to sequence number. The youngest command has to be undone first and therefore function `UndoCommand` will reverse the returned collection and then iterate from start to end, calling `Undo` on each member. In case of redo, `FindDependentCommandsForRedo` will return in descending order according to sequence number. The oldest command has to be redone first

(it certainly cannot depend on any command in the collection), and therefore function `RedoCommand` will reverse the returned collection and then iterate from start to end, calling `Redo` on each member.

That means that weakened stable execution property holds.

□

Stable result property. The same argument as for proof of weakened stable execution property can be used. There is no dependent executed younger command in the `global_history_buffer` during undo of command ⇒ the property holds for undo. There is no older command in the `global_history_buffer` on which requested command to redo depends, command are also undone from younger to older and redone from older to younger ⇒ the property holds for redo. □

Commutative undo property. There are total n commands in the `global_history_buffer` which are randomly distributed into m `history_buffers`. We have two randomly selected commands C_i and C_j.

Undo operation. Both C_i and C_j are executed. There are three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of dependent commands, they can be undone in any order and the result will be the same.
- C_i and C_j are independent but their sets of dependent command intersect each other. Let us call the set of commands in the intersection CS. If C_i is undone prior to C_j, all commands in CS are undone prior to C_i and also prior to C_j. During undo of C_j, commands in CS are already undone, so they will not be selected to undo anymore and only commands which are dependent on C_j but not on C_i will be undone. Similarly for opposite order of undo. Results will be the same, because actions over one particular construct will be always done in one unique order.
- C_i and C_j are dependent, C_j depends C_i. Then during undo of C_i, C_j will be undone prior to C_i and subsequent explicit undo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to undo explicitly by the user before undoing C_i, the situation is similar to the previous one.

Redo operation. Both C_i and C_j are undone. There are three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of commands on which they depend. Then they can be redone in any order and the result will be the same.
- C_i and C_j are independent but their sets of commands on which they depend intersect each other. Let us call the set of commands in the intersection CS. If C_i is redone prior to C_j, all commands in CS are redone prior to C_i and also prior to C_j. During redo of C_j, commands in CS are already redone, so they will not be selected to undo anymore and only commands which are dependent on C_j but not on C_i will be redone. Similarly for opposite

order of redo. Results will be the same, because actions over one particular construct will be always done in one unique order.

- C_i and C_j are dependent, C_j depends C_i. Then during redo of C_i, C_j will be redone prior to C_i and subsequent explicit redo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to redo explicitly by the user before redoing C_i, then the situation is similar to the previous one.

□

6.2.6 Conclusion

The extended the linear undo model is a new model which solves the need of undo in case of environments with several connected history buffers. It is based on linear undo model and adds the dependency search through all command in all workspaces. It can be used in existing systems which use linear undo model with minimal changes in the background undo management.

The main advantage is the fact, that the user with experience with the linear undo does not have to learn a new set of rules, how undo and redo is done. If the user uses only one workspace, the model acts in the same like linear undo. In case of multiple workspaces, undo in the current workspace looks like linear undo and the user is only informed, that also other workspaces were possibly affected.

There is one difference compared to the linear undo model. Linear undo model always brings the document into the state, in which it was once before. The extended linear undo model can possibly create a new document states, which is depicted in Figure 6.2.6.

6.3 Cascade Selective Undo Model

The previous algorithm can serve as a basic undo solution for environments with several stacks.

But, suppose a simple example of two workspaces W1 and W2. In W1 the user creates a construct A, using command C1. Then he/she moves to W2, creates a link to the construct A using command C2 and then continues with his/her work in W2. After some time the user decides that creation of construct A was a mistake and there is no use for it - so he/she decides to undo it. Because the command which created link to A in W2 is buried under many other commands, he/she moves to W1 and calls undo. After this, work in both workspaces is lost, because Extended linear undo model has found a dependency between the creation command in W1 and creation command in W2 - both touches the same construct. This means that C2 must be also undone, but to undo C2, every single command above the C2 has to be undone. Situation is illustrated in Figure6.3.

This situation is extreme and usually this scenario does not mean undoing all work, but even unnecessary undoing parts of the work may be annoying for the user. The fact is, that if the other commands in the buffer W2 are independent of command C2, they do not have to be undone and the system will still stay in stable state (no dead references). This approach would be much more user-friendly and it some situations, it would greatly save user's time.

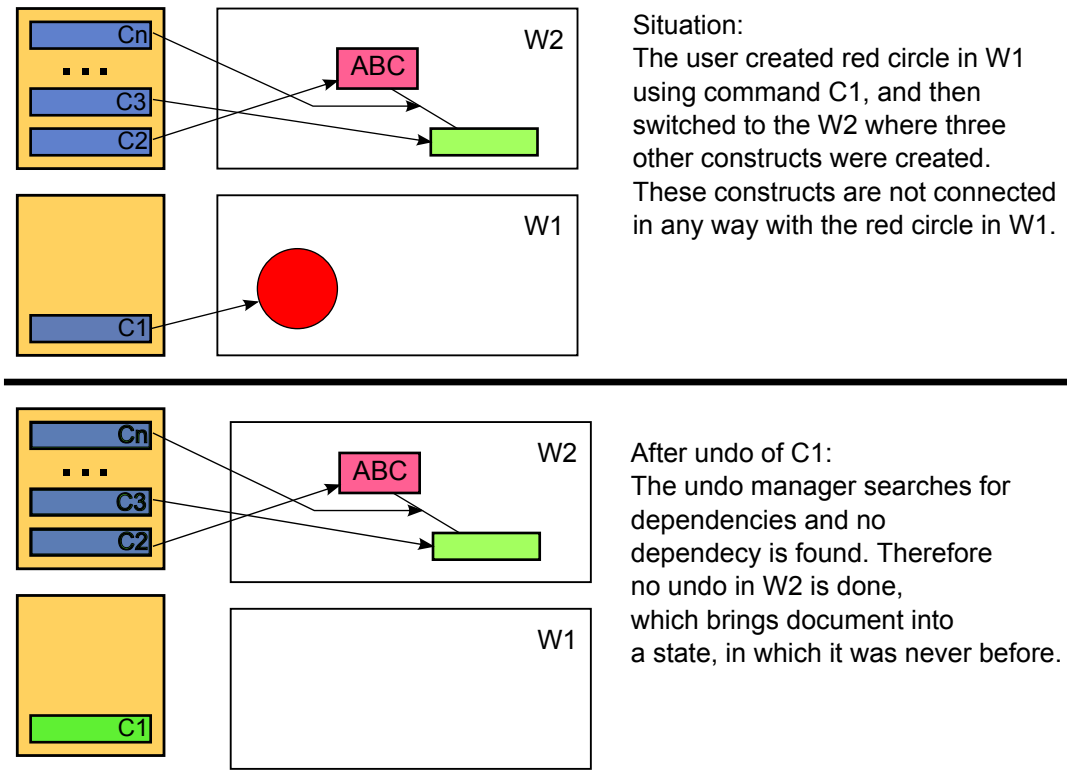


Figure 6.2: Undo in W1 causes creation of new state of document.

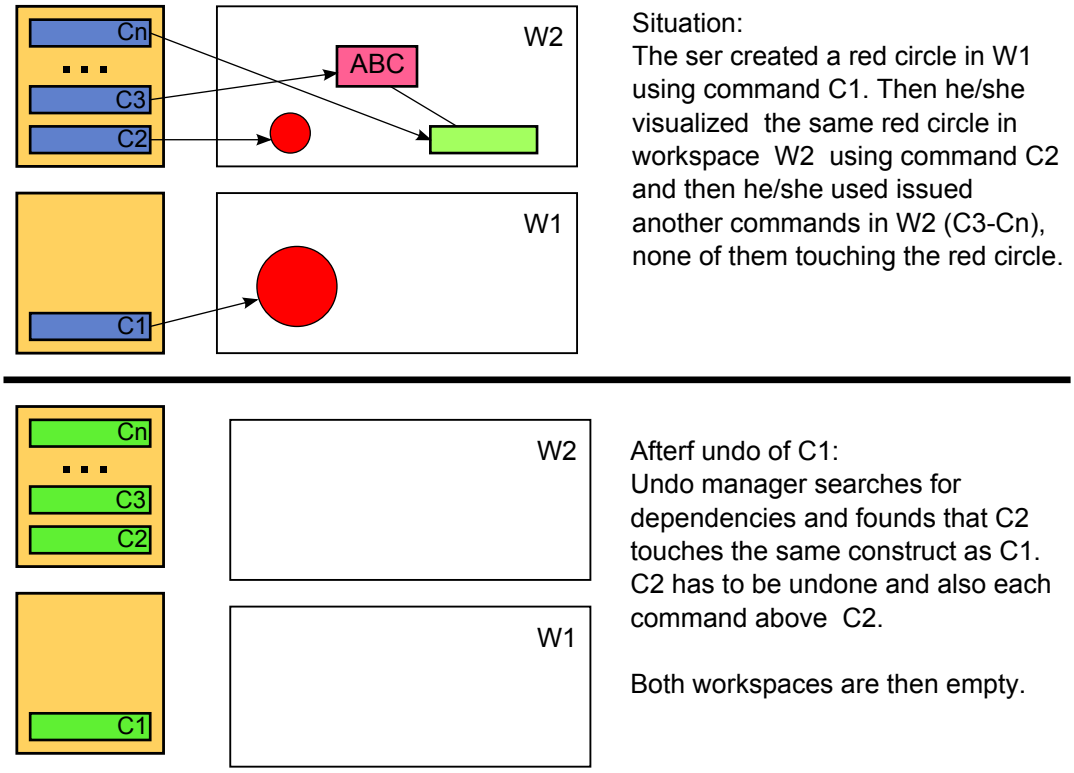


Figure 6.3: Undo in W1 causes undo of all commands in W2.

6.3.1 Requirements

This algorithm should solve the previous situation but not only this. If it is allowed to undo commands in the middle of the stack, when dependency is found, it is not a ptoblem to allow it on the user's direct request ⇒ thn new algorithm should offer the selective undo functionality.

Selective undo implies undoing an redoing commands out of their original context, which means that stable execution property cannot hold true. However, if dependencies are tracked well at least weakened stable execution property can hold true and this is sufficient to keep the document in stable state.

The properties to hold true are:

- **Weakened stable execution property**
- **Stable result property** - Model would not be usable without satisfying this property.
- **Commutative undo property** - Important for intuitive usage.
- **Minimalistic undo property** - Saves user's time.

There is not a big difference as compared with the extended linear undo model. Stable result and commutative undo property should hold true anyway in each command based model, because without them the model could provide unpredictable results. Support of minimalistic undo property is the solution for the situation described in previous chapter.

The behavior of model:

1. Each executed command will be offered to undo.
2. Each undone command will be offered to redo.
3. After successful call of execute operation on command C in workspace W, command C will be executed and it will become the last command in the history buffer belonging to workspace W.
4. After successful call of undo operation on command C in workspace W, command C will be undone.
5. After successful call of redo operation on command C in workspace W, command C will be redone.
6. Commands are never discarded from the history buffer.

This behavior respects analysis provided in Section 5. The user will be able to undo any executed and redo any undone command, no commands are discarded from the buffer during execution of a new command. This function can be voluntary added on direct user's request. It could make history buffer more transparent, when the user is sure that the particular commands are not needed anymore. Removing commands from the stack will not bring the system into unstable state.

6.3.2 Principle and Analysis

The requested behavior differs a lot from the extended linear undo model, but the key principles remains. The command can now be undone or redone in the middle of the stack. This situation exists also in the extended linear undo model, but when there is a request to undo a command, which is not located directly bellow the top of the stack, all commands above it have to be undone. This concept is mandatory, if the stable execution property should hold true.

But as long as only the weakened stable execution property is the goal, this approach can be relaxed and commands which are not dependent on the command being undone/redone, can be left untouched. However the weakened stable execution property says that all dependent commands must be undone/redone prior to the selected command and therefore the dependency search function must be run to construct collection of dependent commands before actual undo or redo is performed. This function can work similar as `FindDependentCommandsForUndo` and `FindDependentCommandsForRedo` in the extended linear undo model, but if a dependent command is found, it should not automatically select rest of the stack for undo or redo.

The functions for execution, undo and redo will not be changed much. The main difference is, that in case of undo and redo, there is no implicit command to be undone or redone, so an additional argument has to be added.

Command

The structure `command` can remain similar to the command structure used in the extended linear undo model and also functions or methods dealing with the commands stay unchanged. This will help the developers to switch from one model to another, because only the internal logic will have to change. The implementation of all commands, which is usually much larger may stay untouched.

History buffer

The structure `history_buffer` itself must be changed. The variable `index_of_top` has no longer meaning if we want to support selective undo. When this variable is removed, history buffer becomes only a storage for commands and it would make sense to make a two-dimensional array of commands, where first index is key of workspace. Because we also want to stay compatible with the previous extended linear undo model, we will still use the structure `history_buffer`, but without the variable `index_of_top`.

```
struct {
        Command command_stack[];
} History_buffer;
```

The functions which manipulate with the `history_buffer` have to be revisited, because there are major changes in the behavior. Firstly, there is now the possibility of having a so-called "hole" in the stack. In the extended linear undo model, there is always a continuous sequence of executed commands at the bottom of the stack, then there is the top of the stack and then continuous sequence of undone commands. But the support of selective undo dictates possibility of "mixing" executed and undone commands. This means, that functions

GetCommandToUndo and GetCommandToRedo are not useful anymore and they have to be replaced with the functions returning specific command from the stack according to the command's position in the stack.

Because there is no top of the stack, also functions MovePointerToTopUp and MovePointerToTopDown have no meaning and they will not be used anymore.

Therefore, command stack becomes rather a list than a classic stack. The functions in the extended linear undo model push commands to the end (top) of the stack and the commands are also removed from the end, according to the LIFO behavior. If commands are never removed from the stack, function Pop is useless. For adding commands, we can use function Append, which will add the executed command to the end of the command_stack collection.

Overview of function manipulating with History_buffer:

- ItemCount(General_collection) - Returns number of items in general collection. Calling on history buffer, it returns number of items in command_stack[].
- Append(General_collection, Item) - Adds one item to the end of the collection.
- GetCommandByPosition(General_collection, Index) - Returns item which resides on the index'th position. If there is no such item, the null value is returned. Index is zero based.
- GetCommandByKey(General_collection, Key) - Returns item with specified Key (command with specified sequence_number. If there is no such item, the null value is returned.

The function portfolio is smaller compared to the extended linear undo model and probably all of these functions can be provided by programming environment, if it supports collection management.

Global History Buffer

The global_history_buffer structure remains also the same as in the extended linear undo model.

6.3.3 Algorithm

The algorithm will have some parts similar to the extended linear undo model - when undo or redo is issued, particular command has to be found in the stack, dependency search will be issued and all dependent commands will be undone or redone.

We start with the modified versions of the helper functions, which search for dependencies - see Algorithms 12 and 13. Both functions are only lighter versions of the same functions from the extended linear undo model.

The ExecuteCommand function (see Algorithm 14) only executes the command and appends it to the end of the buffer. There is no other work to do, because no commands are being discarded during execution.

Algorithm 12 Undo dependencies searcher for cascade selective undo model

```
1:  procedure FindDependentCommandsForUndo(command_to_undo)
2:      index ← GetIndexToGlobalBuffer(command.sequence_number)
3:                                                  ▷ Affected construct keys
4:      ack ← command.keys_of_affected_constructs
5:                                                  ▷ Commands to undo
6:      commands2undo ← command
7:      for i ← index + 1 to ItemCount(global_command_buffer) do
8:          comm ← global_command_buffer[i]
9:          if comm.is_undone then
10:             continue
11:         end if
12:         if ack ∩ comm.keys_of_affected_constructs then
13:             ack ← ack ∪ comm.keys_of_affected_constructs)
14:             commands2undo ← commands2undo ∪ comm
15:         end if
16:     end for
        return commands2undo
17: end procedure
```

Algorithm 13 Redo dependencies searcher for cascade selective undo model

```
1:  procedure FindDependentCommandsForRedo(command_to_redo)
2:      i ← GetIndexToGlobalBuffer(command.sequence_number)
3:                                                  ▷ Affected construct keys
4:      ack ← command.keys_of_affected_constructs
5:                                                  ▷ Commands to redo
6:      commands2redo ← command
7:      for i ← i - 1 downto 0 do
8:          comm ← global_command_buffer[i]
9:          if ¬ comm.is_undone then
10:             Continue
11:         end if
12:         if ack ∩ comm.keys_of_affected_constructs then
13:             ack ← ack ∪ comm.keys_of_affected_constructs)
14:             commands2redo ← commands2redo ∪ comm
15:         end if
16:     end for
        return commands2redo
17: end procedure
```

Functions `UndoCommand` and `RedoCommand` (see Algoritms 15 and 16) are only slightly changes - there is no correction of pointer to the top the stack after the command has been successfuly undone/redone.

Algorithm 14 Execute command in cascade selective undo model

```
1: procedure ExecuteCommand(workspace_key, command)
2:     history_buffer ← GetHistoryBuffer(workspace_key)
3:     if E thenxecute(command)
4:         DiscardRedoCommands(history_buffer)
5:         Push(history_buffer, command)
6:         MoveTopPointerUp(history_buffer,1)
7:         Append(global_history_buffer, command)
8:         return true
9:     end if
10:    return false
11: end procedure
```

Algorithm 15 Undo command in cascade selective undo model

```
1: procedure UndoCommand(workspace_key, com_index)
2:     workspace ← global_workspace_collection[workspace_key]
3:     com2undo ← GetCommandByIndex(workspace_key, com_index)
4:                                                  ▷ Command to undo
5:     if com2undo ≡ NULL then return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo)
8:                                          ▷ Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 16 Redo command in cascade selective undo model

```
1: procedure RedoCommand(workspace_key, com_index)
2:     workspace ← global_workspace_collection[workspace_key]
3:     com2redo ← GetCommandByIndex(workspace_key, com_index)
4:                                                  ▷ Command to redo
5:     if com2redo ≡ NULL then return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo)
8:                                          ▷ Collection of commands to redo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Redo(com2redo_collection[i])
12:    end for
13:    return true
14: end procedure
```

6.3.4 Correctness

This subsection presents proofs, that the cascade selective undo model is correct and gives the required results.

All functions which manipulates with the data structures are either reused from the extended linear undo model and thus their correctness has already been proven or they are considered to be so simple, that they are provided by the implementation of the collection and therefore they are also considered to be correct.

Both functions that searches for dependencies (`FindDependentCommandsFor Undo` and `FindDependentCommandsForRedo`) are slightly changed. In fact, they are only simplified versions of the same functions from the extended linear undo model. The only difference is omitting the construction of `ahb` structure and the subsequent check, whether the inspected command is located in the affected buffer. However, the requested results of functions are different and invariant 1 is not valid anymore, so we need a new proof of correctness, although it will be similar to the proofs in section describing the extended linear undo model.

Theorem 5. *Function `FindDependentCommandsForUndo` will return a collection of commands $Coms_{toUndo}$, which contains all executed commands younger than its argument C_{toUndo}, which are dependent on C_{toUndo}.*

Proof of theorem 5. Suppose, that there is executed command $C_{dependent}$ younger then C_{toUndo}, which is transitively dependent on C_{toUndo} and $C_{dependent}$ is not present in $Coms_{toUndo}$ after successful call of `FindDependentCommandsForUndo`.

There are total n commands in the `global_history_buffer` which are randomly distributed into m `history_buffers`.

The function creates collection `ack` (affected construct keys), initialized with `keys_of_affected_constructs` of C_{toUndo}.Then the function iterates through all commands in `global_history_buffer` from older to younger, starting with direct successor of C_{toUndo}. Each such a command C_i is

- added into collection $Coms_{toUndo}$ AND
- its `keys_of_affected_constructs` collection is merged with `ack` collection

If the set of `keys_of_affected_constructs` of C_i intersects with `ack`.

The condition is true, if and only if C_i affects a construct, which was already touched by one of constructs in $Coms_{toUndo}$. Because $Coms_{toUndo}$ was originally initialized with the `keys_of_affected_constructs` of C_{toUndo}, C_i is transitively dependent on C_{toUndo}.

But it means, that command $C_{dependent}$ may not exist, because it would be selected to undo. □

Theorem 6. *Function `FindDependentCommandsForRedo` will return a collection of commands $Coms_{toRedo}$, which contains all undone commands older than its argument C_{toRedo} on which C_{toRedo} depends.*

Proof of theorem 6. Suppose, that there is undone command $C_{dependent}$ older then C_{toRedo}, on which C_{toRedo} (transitively) depends and $C_{dependent}$ is not present in $Coms_{toRedo}$ after successful call of `FindDependentCommandsForRedo`.

The function creates collection `ack` (affected construct keys), initialized with `keys_of_affected_constructs` of C_{toRedo}. Then the function iterates through all commands in `global_history_buffer` from younger to older, starting with direct predecessor of C_{toRedo}. Each such a command C_i is

- add into collection $Coms_{toRedo}$ AND
- its `keys_of_affected_constructs` collection is merged with `ack` collection

If the set of `keys_of_affected_constructs` of C_i intersects with `ack`.

The condition is true, if and only if C_i is affected by a construct, which was already affected by one of the commands in $Coms_{toRedo}$. Because the $Coms_{toRedo}$ was originally initialized with the `keys_of_affected_constructs` of C_{toRedo} the C_{toRedo} is transitively dependent on C_i.

The second condition means, that if the key of history buffer H_i, to which C_i belongs, is already in `ahb` collection, a command older than C_i from the H_i has already been selected selected for redo.

But that means, that command $C_{dependent}$ may exist, because it would be selected to redo. □

Behavior

The requirements section specifies six requirements for behavior of cascade selective undo model. The following enumeration proves each of these requirements to be met:

1. The model allows any executed command to be selected for undo - there is no function, which would restrict the commands to be selected for undo.
2. The model allows any undone command to be selected for redo - there is no function, which would restrict the commands to be selected for redo.
3. Function `ExecuteCommand` always executes the command $C_{toExecute}$ and if the execution of $C_{toExecute}$ is successful, it appends $C_{toExecute}$ to the end of the history buffer belonging to the workspace in which $C_{toExecute}$ has been executed.
4. Function `UndoCommand` undoes command C_{toUndo} and all commands dependent on C_{toUndo}. Dependent commands are gathered by calling `Find DependentCommandsForUndo` with C_{toUndo} as an argument. As it has been proven, function `FindDependentCommandsForUndo` always returns a collection containing command in the argument and all younger executed dependent commands. `UndoCommand` then calls function `Undo` on every item from this collection, which means, that C_{toUndo} will be undone.
5. Function `RedoCommand` redoes command C_{toRedo} and all commands on which C_{toRedo} depends. Dependent commands are gathered by calling `Find DependentCommands` with C_{toUndo} as an argument. As it has been proven, function `FindDependentCommandsFor` always returns a collection containing C_{toRedo} and all older undone commands on which C_{toRedo} depends . `RedoCommand` then calls function `Redo` on every item from this collection, which means, that C_{toRedo} will be redone.

6. There is no function in the whole model, which would remove a command from the `history_buffer`. Thus, commands are never discarded from the stack.

Properties

Weakened Stable Execution Property. It has already been proven, that the function `FindDependentCommandsForUndo` will return all younger commands dependent on the command passed to it as an argument and similarly, `FindDependent CommandsForRedo` will return all younger commands on which the command passed to it as an argument depends.

Function `UndoCommand` will call `FindDependentCommandsForUndo` to get the collection of all dependent commands. The collection is ordered in ascending order according to the sequence number of the commands. The youngest command has to be undone first and therefore `UndoCommand` reverses the returned collection and then the function iterates over it from start to end, calling `Undo` on each member.

Function `RedoCommand` will call `FindDependentCommandsForRedo` to get the collection of all dependent commands. The collection is ordered in descending order according to the sequence number of the commands. The oldest command has to be undone first and therefore `RedoCommand` reverses the returned collection and then the function iterates over it from start to end, calling `Redo` on each member.

That means that the weakened stable execution property holds.

□

Stable result property. The proof has to be done for both operations, undo and redo.

When undo operation is invoked on command C_{toUndo}, function `UndoCommand` is called and C_{toUndo} is passed as an argument to it. As it has been proven in proof of weakened stable execution property, all executed commands dependent on C_{toUndo} will be undone prior to C_{toUndo} and they are undone in the order from the youngest to the oldest. This means, that the property holds true.

Similarly, when redo operation is performed on command C_{toRedo}, function `RedoCommand` is called and C_{toRedo} is passed as an argument to it. As it has been proved in proof of weakened stable execution property, all commands on which C_{toRedo} depends are redone prior to C_{toRedo} in the order from the oldest to the youngest. This means, that the property holds true.

□

Commutative undo property. There are total n commands in the `global_history _buffer` which are randomly distributed into m `history_buffers`. We have two randomly selected commands C_i and C_j.

Undo operation. Both C_i and C_j are executed. There are three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of dependent commands. They can be undone in any order and the result will be the same.

- C_i and C_j are independent but their sets of dependent commands intersect each other. Let us call the set of commands in the intersection CS. If C_i is undone prior to C_j, all commands in CS are undone prior to C_i and also prior to C_j. During undo of C_j, the commands in CS are already undone, so they will not be selected to undo once mode and only commands which are dependent on C_j but not on C_i will be undone. Similarly for the opposite order of undo. The results will be the same, because actions over one particular construct will be always performed in one unique order.

- C_i and C_j are dependent, C_j depends C_i. Then during undo of C_i, C_j will be undone prior to C_i and the subsequent explicit undo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to undo explicitly by the user before undoing C_i, then the situation is similar to the previous one.

Redo operation. Both C_i and C_j are undone. There are again three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of commands on which they depend. Then they can be redone in any order and the result will be the same.

- C_i and C_j are independent but their sets of commands on which they depend intersect each other. Let us call the set of commands in the intersection CS. If C_i is redone prior to C_j, all commands in CS are redone prior to C_i and also prior to C_j. During redo of C_j, the commands in CS are already redone, so they will not be selected to undo once more and only commands which are dependent on C_j but not on C_i will be redone. Similarly for the opposite order of the redo. Results will be the same, because actions over one particular construct will be always done in one unique order.

- C_i and C_j are dependent, C_j depends C_i. Then during redo of C_i, C_j will be redone prior to C_i and subsequent explicit redo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to redo explicitly by the user before redoing C_i, then the situation is similar to the previous one.

□

Minimalistic undo property. We would like to prove, that for a randomly selected command C_{toUndo}, only C_{toUndo} and the commands which depend on C_{toUndo} will be undone. Similarly for randomly selected command to redo C_{toRedo}, only C_{toRedo} and the commands on which C_{toRedo} depends will be redone.

To undo a command C_{toUndo}, function `UndoCommand` with C_{toUndo} as an argument is called. This function will construct collection $Coms_{toUndo}$ by calling function `FindDependentCommandsForUndo`, passing C_{toUndo} as an argument to it. It has already been proven, that `FindDependentCommandsForUndo` will return collection, which contains only C_{toUndo} and commands dependent on C_{toUndo}. Function `Undo` is then called on each member $Coms_{toUndo}$ $\Rightarrow$ minimalistic undo property holds for undo operation.

To redo command C_{toRedo}, function `RedoCommand` with C_{toRedo} as an argument is called. This function will construct collection $Coms_{toRedo}$ by calling function `FindDependentCommandsForRedo`, passing C_{toRedo} as an argument to it. It has already been proven, that `FindDependentCommandsForRedo` will return collection, which contains only C_{toRedo} and all commands on which C_{toRedo} depends. Function `Redo` is then called on each member of $Coms_{toRedo} \Rightarrow$ minimalistic undo property holds for redo operation.

□

6.3.5 Conclusion

The presented model is a new undo model, which brings the possibility of undoing any command at any time in the environments with multiple connected stacks. It suits very well to these environments, because it minimizes the number of commands, which have to be undone/redone because of dependency between two commands - the commands which are not dependent on the commands being undone or redone are not affected. It allows more states of the document to be reached just by using undo and redo in comparison with the extended linear undo model.

The implementation of the model is not difficult, because the model can reuse commands from the extended linear undo model. The internal functions also share the common code with the extended linear undo model, functions are generally even more simple. This surprising property results from the fact, that this model is not aimed to be backwards compatible with the linear undo model and therefore many unnecessary features can be omitted.

6.4 Combined Undo Model

The cascade selective undo model brings possibility of undoing any command at any time. But there is one situation, which it handles badly - when the user wants to get the whole document (all the workspaces) to some specific state (so called *global linear undo* or), which could once exist in the past after the execution of particular command. If there are not many dependencies among the stacks, the user has to undo each command by hand. This can be annoying for the user, especially if the requested command, which serves as the marker of the requested state, is very old and it is deeply nested in the bottom of a stack.

The extended linear undo model behaves better in this situation, but the result also depends on the number of dependencies among commands. The advantage is (if the user interface supports it), that the user can usually select a command in the middle of the buffer and all the commands above it will be undone. Bringing the document to a specific state then means at least one click in each history buffer. This is significantly better in comparison with the cascade selective undo model (the number of the manual undo operations may be as high as the number of younger commands), but the ideal solution would be just one special undo action, which would bring the document into the desired state. The other problem is, that the user has to know the global order of commands - which command is the one above the requested command - in all stacks.

Cascade selective undo model is also not so intuitive as the extended linear undo model. The users are usually used to use undo and redo buttons or keyboard shortcuts Ctlr+Z, Ctrl+Y, which are present in many applications supporting undo and redo. But such interface cannot be used in the cascade selective undo model, because there are no implicit commands to undo or redo - the user has to select the particular commands from a list or stack.

The idea of combined undo model is to take the best from the both previously presented models and extend them by adding a possibility of bringing the document into a specific state. It should combine the intuitiveness of the extended linear undo model with the possibilities that offer the cascade selective undo.

6.4.1 Requirements

The requirements for the combined undo model results from both previous models.

The properties to hold true are:

- **Weakened stable execution property**
- **Stable result property** - The model would not be usable without satisfying this property.
- **Commutative undo property** - Important for intuitive usage.
- **Minimalistic undo property**- Saves user's time.

The set of properties is identical to cascade selective undo model. The stable execution property cannot hold true, if we want to offer any command to undo or redo and simultaneously obey the minimalistic undo property.

We also would like to offer a possibility of performing undo in the traditional way (undo and redo button), which implies implementation of mechanism which will select one command to undo and redo in each stack.

The behavior of model should be as follows:

1. Each executed command can be undone.
2. Each undone command can be redone.
3. The youngest executed command will be offered for linear undo. When such a command does not exist, the linear undo is not possible.
4. If there is a continuous sequence of undone commands at the end of the history buffer, the oldest command from this sequence will be offered for linear redo.
5. Each executed command can be selected for global linear undo. After performing global linear undo on command C_{toUndo}, there will be no younger executed command than C_{toUndo} and no older undone command than C_{toUndo} in any history buffer in the document.
6. After a successful call of execute operation on command $C_{toExecute}$ in workspace W, command $C_{toExecute}$ will be executed and it will become the last command in the history buffer belonging to W.

7. After a successful call of undo operation on command C_{toUndo} in workspace W, C_{toUndo} will be undone.

8. After a successful call of redo operation on command C_{toRedo} in workspace W, C_{toRedo} will be redone.

9. Commands are never discarded from the history buffer.

6.4.2 Analysis

The combined undo model is a mix of the extended linear undo model and cascade selective undo model, which are both already described and proven to be correct. The purpose of this chapter is to find the way how to combine their functions in one model.

As mentioned in requirements section, if we want to allow a model to act like the linear undo model, there must be a mechanism which selects which commands will be undone or redone when the undo or the redo button is pressed. Extended linear undo model uses a pointer to the top of the stack, where the border between undone and executed commands lies. But if we support selective undo and therefore commands in the middle of the stack may be undone, there can be more than one such a border. For this purpose, we create a pointer to so-called*virtual stack top*. This pointer always points one position above the youngest executed command. It can be easily computed on the fly by iterating through history buffer starting at the end. The command to undo will be the one bellow the virtual stack top, the command to redo will be the one to which the pointer points.

Global linear undo is a new feature, which is not present in the extended linear undo model, nor in cascade selective undo model. It is called "undo", but it may also perform redo operations if it is necessary - all undone commands older than the selected command have to be redone and undo all executed commands younger than the selected command have to be undone. Implementation can be easily done by one iteration through global history buffer, which will be introduced later in this section.

It should be noted, that the state reached after performing global linear undo may not be the exact state, in which document was after the original execution of the selected command. If there were any undone commands present in any stack during the original execution, the global linear undo will redo these commands. Saving the exact state of stacks after execution of each command cannot be done - it would be a new linear undo model (based on saving document state after each step).

Execution of a new command would be done in the way that selective undo requires - commands to redo are not discarded. It would be possible to discard all commands above the virtual stack, but it would collide with definition of the selective undo.

Command

The `Command` structure can be again taken from the extended linear undo model. It carries all necessary information to build up a hierarchy of dependencies.

The functions which operate with the structure - `Execute`, `Undo` and `Redo` - also remain and their semantics is unchanged.

History buffer

The history buffer can be used form cascade selective undo model. As the pointer to the top of the stack will be computed on the fly, we do not need a variable to store this information, which makes this implementation is sufficient.

```
struct {
        Command command_stack[];
        int index_of_top;
} History_buffer;
```

But the functions which manipulate with the history buffer are changed, because we would like to support also the linear undo:

- `ItemCount(General_collection)` - Returns the number of items in a general collection.
- `Append(General_collection, Item)` - Adds one item at the end of the collection.
- `GetCommandByPosition(General_collection, Index)` - Returns the item which resides on the index'th position of the collection. If there is no such item, the null value is returned. Index is zero based.
- `GetCommandByKey(General_collection, Key)` - Returns the item with specified Key (in case of a command with specified `sequence_number`). If there is no such item, the null value is returned.
- `GetVirtualStackTop(General_collection)` - Returns the index of the virtual top of the stack.
- `GetCommandToLinearUndo(History_buffer)` - Returns the command, which is right under the virtual top of the stack or null value, if such a command does not exist.
- `GetCommandToLinearRedo(History_buffer)` - Returns the command on which points the virtual top of the stack or the null value, if such a command does not exist.

All functions except `GetVirtualStackTop` are known from the previous algorithms. `GetCommandToLinearUndo` and `GetCommandToLinearRedo` work in the same way as `GetCommandToUndo` and `GetCommandToRedo` from the extended linear undo algorithm, but instead of using `pointer_to_top` they get the correct index of the stack of the top by calling `GetVirtualStackTop`.

Algorithm 17 shows implementation of `GetVirtualStackTop`.

Global history buffer

Global history buffer can remain unchanged, as it again serves only as a simple collection for command ordering according to their sequence number.

Algorithm 17 Get index of the virtual stack top

```
1: procedure GetVirtualStackTop(workspace_key)
2:     hist_buff ← GetHistoryBuffer(workspace_key)
3:     top_index ← ItemCount(hist_buff.command_stack)
4:     for i ← ItemCount(hist_buff.command_stack) - 1 to 0 do
5:         if hist_buff.command_stack[i].is_undone ≡ false then
6:             break
7:         end if
8:         top_index ← top_index - 1
9:     end for
10:    return top_index
11: end procedure
```

6.4.3 Algorithm

The algorithm has to support three different types of undo and redo:

- **Linear** - Acts in the same way as the extended linear undo model.
- **Selective** - Acts in the same way as cascade selective undo model.
- **Global** - Gets the document into a state in which it once was after the original execution of the selected command.

The functions for dependency search (see Algorithms 18 19) are well-known and they have not been changed much.

Both functions are parametrized by one boolean variable `is_linear`, which specifies, whether the function searches for the dependencies as the extended linear undo model requires (if true) or as the cascade selective undo model requires (if false). Both returns a collection of commands, which should be reversed and then undo or redo may be performed on each member of the collection.

The xext important function is `Execute` (see Algorithm 20), which executes a new command and appends it to the history buffer. Because execution is done in the same way as the selective undo requires, function is similar to `Execute` function used in the cascade selective undo model.

There are together four functions for undo and redo - `UndoCommandSelective` (Algorithm 21), `RedoCommandSelective` (Algorithm 23), `UndoCommandLinear` (Algorithm 22) and `RedoCommandLinear` (Algorithm 24). Both selective and linear functions take different number of arguments, so they are really presented as four distinct functions, although in real implementation, they could be merged together to avoid unnecessary code duplication - some parts of the code are shared.

The last function which allows the user to undo and redo commands is called `UndoCommandGlobal` (see Algorithm 25). As mentioned before, the purpose of this function is to get document into a state, when there are no younger executed and no older undone commands in the whole document than selected command.

Global history buffer is a perfect structure for the implementation of such a function. The basic idea is an iteration through the buffer, which build up two collections - commands to redo and commands to undo. Each undone younger command goes to the commands to redo, each older executed command

Algorithm 18 Undo dependencies searcher for combined undo model

```
 1: procedure FindDependentCommandsForUndo(command, is_linear)
 2:     index ← GetIndexToGlobalBuffer(command.sequence_number)
 3:                                                  ▷ Affected construct keys
 4:     ack ← command.keys_of_affected_constructs
 5:                                                  ▷ Affected history buffers
 6:     ahb ← GetHistoryBufferKey(command)
 7:                                                  ▷ Commands to undo
 8:     commands2undo ← command
 9:     for i = index + 1 to ItemCount(global_command_buffer) do
10:         comm ← global_command_buffer[i]
11:         if comm.is_undone then
12:             continue
13:         end if
14:         if is_linear ∨ ( ahb ∩ GetHistoryBufferKey(comm)) then
15:             ack ← ack ∪ comm.keys_of_affected_constructs)
16:             commands2undo ∪ commands2undo ∪ command
17:         else if ack ∩ comm.keys_of_affected_constructs then
18:             ack ← ack ∪ comm.keys_of_affected_constructs)
19:             if is_linear then
20:                 ahb ← ahb ∪ GetHistoryBufferKey(comm)
21:             end if
22:             commands2undo ← commands2undo ∪ command
23:         end if
24:     end for
        return commands2undo
25: end procedure
```

to commands to undo. No other dependencies have to be searched, because if there are some dependent commands to undo or redo, they will be surely also undone or redone.

Two final for loops can be switched, because in case of redo, we start with the oldest undone command (which implies that no older dependent undone command exists) and in case of undo we start with the youngest executed command (which similarly implies that no younger dependent executed command exists).

6.4.4 Correctness

Combined undo model is the mix of the extended undo model and cascade undo model, but there are also several new features - virtual stack top and global linear undo.

The `command` structure and functions that belong to it (`Execute`, `Undo` and `Redo`) were reused from the previous algorithms and their correctness has already been proven.

Also the `history_buffer` structure is taken from the cascade selective undo model and it remains without changes and the functions `ItemCount`, `Append`, `GetCommandByPosition` and `GetCommandByKey` as well. Functions `GetCommand ToLinearUndo` and `GetCommandToLinearRedo` are implemented in the same way

Algorithm 19 Redo dependencies searcher for combined undo model

```
1: procedure FindDependentCommandsForRedo(command, is_linear)
2:     i ← GetIndexToGlobalBuffer(command.sequence_number)
3:                                                  ▷ Affected construct keys
4:     ack ← command.keys_of_affected_constructs
5:                                                  ▷ Affected history buffers
6:     ahb ← GetHistoryBufferKey(command)
7:                                                  ▷ Commands to redo
8:     commands2redo ← command
9:     for i = i - 1 downto 0 do
10:        comm ← global_command_buffer[i]
11:        if ¬ comm.is_undone then
12:            continue
13:        end if
14:        if is_linear ∨( ahb ∩ GetHistoryBufferKey(comm)) then
15:            ack ← ack ∪ comm.keys_of_affected_constructs)
16:            commands2redo ← commands2redo ∪ command
17:        else if ack ∩ comm.keys_of_affected_constructs then
18:            ack ← ack ∪ comm.keys_of_affected_constructs)
19:            if is_linear then
20:                ahb ← ahb ∪ GetHistoryBufferKey(comm))
21:            end if
22:            commands2redo ← commands2redo ∪ command
23:        end if
24:    end for
       return commands2redo
25: end procedure
```

Algorithm 20 Execute command in combined undo model

```
1: procedure ExecuteCommand(workspace_key, command)
2:     history_buffer ← GetHistoryBuffer(workspace_key)
3:     if Execute(command) then
4:         Append(history_buffer, command)
5:         Append(global_history_buffer, command)
6:         return true
7:     end if
8:     return false
9: end procedure
```

as `GetCommandToUndo` and `GetCommandToRedo` from the extended linear undo algorithm, but instead of using `pointer_to_top` they get index of stack top by calling `GetVirtualStackTop`. Therefore both functions can be considered as correct as long as `GetVirtualStackTop` returns the correct index of the virtual stack top.

Theorem 7. *Function `GetVirtualStackTop` returns the index of the youngest executed command in the history buffer increased by one. If there is no executed command in the history buffer, the function returns zero.*

Algorithm 21 Selective undo command in combined undo model

```
1: procedure UndoCommandSelective(workspace_key, com_index)
2:     com2undo ← GetCommandByIndex(workspace_key, com_index)
3:                                                  ▷ Command to undo
4:     if com2undo ≡ NULL then
5:         return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo,
   false)
8:                                          ▷ Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 22 Linear undo command in combined undo model

```
1: procedure UndoCommandLinear(workspace_key)
2:     com2undo ← GetCommandToUndo(workspace_key)
3:                                                  ▷ Command to undo
4:     if com2undo ≡ NULL then
5:         return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo,
   true)
8:                                          ▷ Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Proof of theorem7. The function sets the value of `top_index` to the number of items in the `command_stack` of the history buffer. Then it iterates over all commands in the `command_stack` from the youngest to the oldest, in each iteration the value of the `top_index` is decreased by one. The iteration is stopped when the command at the current position is not undone or the bottom of the stack has been reached.

There are total n commands in the history buffer, index is zero based. If the last command in the buffer (position $n-1$) is not undone, the value of `top_index` is not decreased even once and iteration is immediately stopped. Thus, value n is returned, which is correct.

If the youngest executed command in the history buffer is at the position i, $i > 0$ and $i < n$, total $(n-i+1)$ decrementations of the `top_index` are performed before the iteration is stopped. Value $n - (n - i + 1) = i + 1$ is returned, which

Algorithm 23 Selective redo command in combined undo model

```
1: procedure RedoCommandSelective(workspace_key, com_index)
2:     com2redo ← GetCommandByIndex(workspace_key, com_index)
3:                                                  ▷ Command to redo
4:     if com2redo ≡ NULL then
5:         return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo,
   false)
8:                                          ▷ Collection of commands to undo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Redo(com2redo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 24 Linear redo of command in combined undo model

```
1: procedure RedoCommandLinear(workspace_key)
2:     com2redo ← GetCommandToRedo(workspace)
3:                                                  ▷ Command to redo
4:     if com2redo ≡ NULL then
5:         return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo,
   true)
8:                                          ▷ Collection of commands to redo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Redo(com2redo_collection[i])
12:    end for
13:    return true
14: end procedure
```

is correct.

If there is no executed command in the history buffer, total n decrementations of the `top_index` are performed and the value zero is returned, which is correct.

If $n = 0$ (buffer is empty), value of `top_index` is set to a zero (number of items in the `command_stack`), no iteration is done and therefore a zero is returned, which is correct.

The function will always return a correct value and therefore it is correct.

□

Algorithm

Functions which search for dependencies among commands (`FindDependentCommands ForUndo` and `FindDependentCommandsForRedo`) are just mix of the same functions implemted in the previous algorithms. In fact, we could use thee versions

Algorithm 25 Global linear undo

```
 1: procedure UndoCommandGlobal(workspace_key, com_index)
 2:     selected_comm ← GetCommandByIndex (workspace_key, com_index)
 3:     if selected_com ≡ NULL then
 4:         return false
 5:     end if
 6:     i ← GetIndexToGlobalBuffer (selected_comm.sequence_number)
 7:     comm2redo ← empty_collection
 8:     for i ← 0 to index - 1  do
 9:         comm ← global_command_buffer[i]
10:         if comm.is_undone then
11:             comm2redo ← comm2redo ∪ comm
12:         else
13:             continue
14:         end if
15:     end for
16:     comm2undo ← empty_collection
17:     for i ← ItemCount(global_command_buffer) downto index  do
18:         comm ← global_command_buffer[i]
19:         if comm.is_undone then State comm2undo ← comm2undo ∪ comm
20:         else
21:             continue
22:         end if
23:     end for
24:     for  i ← 0 to ItemsCount(comm2redo) do
25:         Redo(com2redo[i])
26:     end for
27:     for  i ← 0 to ItemsCount(comm2undo) do
28:         Undo(com2undo[i])
29:     end for
         State return true
30: end procedure
```

from the extended linear undo model for the linear undo and the versions from cascade selective undo model for the selective undo. Underlaying data structures were changed and therefore, they can be considered as correct.

Function `Execute` is similar to the same function defined for cascade selective undo model and functions `UndoCommandSelective`, `UndoCommandLinear`, `RedoCommandSelective` and `RedoCommandLinear` have all been already defined and proven correct either in the extended linear undo model or in cascade selective undo model.

The only new function is `UndoCommandGlobal`.

Theorem 8. *After successful call of function* `UndoCommandGlobal` *with* C_{toUndo} *as an argument, all commands in the* `global_history_buffer` *younger then the* C_{toUndo} *will be in the state undone and all commands in the* `global_history_buffer` *older than the* C_{toUndo} *will be in the state executed.*

Proof of theorem 8. Suppose, that there are two commands C_{young} and C_{old}. C_{young}

is younger than C_{toUndo}, C_{old} is older than C_{toUndo}. After the successful call of `UndoCommandGlobal`, C_{young} is not in the state undone and C_{old} is not in the state executed.

There are total n commands in the `global_history_buffer`, C_{toUndo} is at the position i, $i \geq 0$ and $i \leq n$ (index is zero based). The function execution has two independent phases:

- During the first phase, the function iterates through all commands younger than C_{toUndo}, starting with the youngest one and stops at position i. Each command, which is not in the state undone, is undone. Since commands are taken from the younger to older, it is not possible, than any command being undone would have any older commands dependent on it - they would be undone before it. All commands younger than C_{toUndo} are undone and therefore command C_{young} may not exist.

- During the second phase, the function iterates through all commands older than C_{toUndo}, starting with the oldest and stops at position $i - 1$. Each command, which is not in the state executed, is redone. Since commands are taken from the older to younger, it is not possible, than any command being redone could be dependent on non-executed command - all commands older than it would be redone before it. All commands older than C_{toUndo} are executed and therefore command C_{old} cannot exist.

□

Behavior

The requirement section specifies nine points, how the model should behave. The following enumeration proves each of these requirements to be satisfied:

1. The model allows any executed command to be selected for the selective undo - there is no function, which would restrict commands to be selected for undo.

2. The model allows any undone command to be selected for selective redo - there is no function, which would restrict commands to be selected for redo.

3. It has been proven, that function `GetCommandToLinearUndo` selects the youngest executed command and offers it to undo.

4. It has been proved, that function `GetCommandToLinearRedo` selects the oldest executed command from the continuous sequence of undone commands at the end of the history buffer and offers it to redo.

5. There is no restriction for selecting a command for global linear undo. It has been proven, that function `UndoCommandGlobal` will get all commands younger than its argument into the state undone and all commands older than its argument into the state executed.

6. Function `ExecuteCommand` executes a command and appends it at the end of the history buffer.

7. Both functions `UndoCommandLinear` and `UndoCommandSelective` call function `FindDependentCommandsForUndo` with C_{toUndo} as an argument. As it has been proven that function `FindDependentCommandsForUndo` will always return a collection containing C_{toUndo} and all executed commands, which depends on C_{toUndo}. Both functions `UndoCommandLinear` and `UndoCommand Selective` then call function `Undo` on every item of the collection, which means, that C_{toUndo} will be undone.

8. Both functions `RedoCommandLinear` and `RedoCommandSelective` call function `FindDependentCommandsForRedo` with C_{toRedo} as an argument. As it has been proved, function `FindDependentCommandsForRedo` will always return a collection containing C_{toRedo} and all older undone commands on which cC_{toRedo} depends. Both functions `RedoCommandLinear` and `Redo CommandSelective` then call function `Redo` on every item from this collection, which means that C_{toRedo} will be redone.

9. There is no function in the whole model, which would remove command from the `histrory_buffer`. Thus, commands are never discarded from the stack.

Properties

There are total four properties claimed to hold true:

Weakened stable execution property. It has already been proved, that function `FindDependentCommandsForUndo` will return all younger commands dependent on the command passed to it as an argument and similarly, `FindDependent CommandsForRedo` will return all younger commands on which command passed to it as an argument depends. This behavior is independent of the value of the second argument.

For undo of command C_{toUndo} either function `UndoCommandLinear` or `Undo CommandSelective` is called. Both call `FindDependentCommandsForUndo` to get the collection of commands, which depend on C_{toUndo}. The collection is ordered in the ascending order according to the sequence number of commands. The youngest command has to be undone first and therefore the both functions reverse the order of the collection and then iterate from start to end, calling `Undo` on each member. Commands are undone from the youngest to the oldest and no command, which depends on C_{toUndo} is omitted $\Rightarrow$ weakened stable execution property holds true.

For redo of command C_{toRedo} either function `RedoCommandLinear` or `Redo CommandSelective` is called. Both call `FindDependentCommands ForRedo` to get the collection of commands, on which depends the C_{toRedo}. The collection is ordered in the descending order according to the sequence number of commands. The oldest command has to be redone first and therefore the both functions reverse the order of the collection and then iterate from start to end, calling `Redo` on each member. Commands are redone from the youngest to the oldest and no command, on which C_{toRedo} depends is omitted $\Rightarrow$ weakened stable execution property holds true.

Function `UndoCommandGlobal` undoes all executed commands younger then the command passed as an argument and redoes all undone commands older

than the argument. It has been proven, that it is performed in the order, that satisfies weakened stable execution property.

That means that weakened stable execution property holds.

□

Stable result property. The proof has to be done for all three operations: undo, redo and global linear undo.

When an undo operation is invoked on command C_{toUndo}, function `UndoCommandLinear` or `UndoCommandSelective` is called and C_{toUndo} is passed as an argument to it. As it has been proved in the proof of weakened stable execution property, all executed commands dependent on C_{toUndo} and younger than the C_{toUndo} will be undone prior to the C_{toUndo} and they will be undone in the order from the youngest to the oldest ⇒ the stable result property holds true.

When a redo operation is invoked on command C_{toRedo}, function `RedoCommandLinear` or `RedoCommandSelective` is called. As it has been proven in the proof of weakened stable execution property, all commands older than C_{toRedo} on which C_{toRedo} depends will be redone prior to the C_{toRedo} and they will redone in the order from the oldest to the youngest ⇒ the stable result property holds true.

When a global linear undo is performed, function `UndoCommandGlobal` is called and it undoes all executed commands younger then the command passed to it as an argument and redoes all undone commands older than the argument. Undo is performed in the order from the youngest to the oldest, redo in the order from the oldest to the youngest. This means, that the property still holds true. □

Commutative undo property. There are total n commands in the `global_history_buffer` which are randomly distributed into m `history_buffers`. We have two randomly selected commands C_i and C_j.

Undo operation. Both C_i and C_j are executed. There are three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of dependent commands. Then they can be undone in any order and the result will be the same.

- C_i and C_j are independent but their sets of dependent commands intersect each other. Let us call the set of the commands in the intersection CS. If C_i is undone prior to C_j, all commands in CS are undone prior to C_i and also prior to C_j. During undo of C_j, commands in CS are already undone, so they will not be selected for undo once more and only commands which are dependent on C_j but not on C_i will be undone. Similarly for the opposite order of undo. Results will be the same, because actions performed on the one construct will be always done in the one unique order.

- C_i and C_j are dependent, C_j depends C_i. Then during undo of C_i, C_j will be undone prior to C_i and subsequent explicit undo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to undo explicitly by the user before undoing C_i, then the situation is similar to the previous one.

Redo operation. Both C_i and C_j are undone. There are three situations, which have to be handled:

- C_i and C_j are independent and there is no intersection between their sets of commands on which they depend. Then they can be redone in any order and result will be the same.
- C_i and C_j are independent but their sets of commands on which they depend intersect each other. Let us call the set of the commands in the intersection CS. If C_i is redone prior to C_j, all commands in CS are redone prior to C_i and also prior to C_j. During redo of C_j, commands in CS are already redone, so they will not be selected to undo anymore and only commands which are dependent on C_j but not on C_i will be redone. Similarly for opposite order of redo. Results will be the same, because actions on the one construct will be always performed in one unique order.
- C_i and C_j are dependent, C_j depends C_i. Then during redo of C_i, C_j will be redone prior to C_i and subsequent explicit undo of C_j will not be possible, which satisfies commutative undo property. If C_j will be selected to redo explicitly by the user before undoing C_i, then the situation is similar to the previous situation.

Global linear undo.
If both previous operations satisfy commutative undo property, the global linear undo cannot break commutative undo property either, because it will always get document into a state, in which it would be if linear undo was used. Linear undo satisfies commutative undo property and therefore, global linear undo too.

□

Minimalistic undo property. Minimalistic undo property holds only for selective undo.

We would like to prove, that for a randomly selected command for selective undo C_{toUndo}, only C_{toUndo} and the commands younger than C_{toUndo} which depends on C_{toUndo} will be undone. Similarly for a randomly selected command to selective redo C_{toRedo}, only C_{toRedo} and the commands older than C_{toRedo} on which C_{toRedo} depends will be redone.

To undo a command C_{toUndo}, function `UndoCommandSelective` with C_{toUndo} as the argument is called. This function will construct a collection of commands to undo by calling function `FindDependentCommandsForUndo`, passing C_{toUndo} boolean value "false" as the arguments to it. It has already been proven, that `FindDependentCommandsForUndo` will return the collection, which contains C_{toUndo} and all executed command dependent on C_{toUndo}. Function `Undo` is then called on each member of the collection. Only C_{toUndo} and executed commands dependent on C_{toUndo} are undone ⇒ the property holds true.

To redo a command C_{toRedo}, function `RedoCommandSelective` with C_{toRedo} as the argument is called. This function will construct a collection of commands to redo by calling function `FindDependentCommandsForRedo`, passing C_{toRedo} and boolean value "false" as the arguments to it. It has already been proven, that `FindDependentCommandsForRedo` will return the collection, which contains

C_{toRedo} and all commands older than C_{toRedo} on which C_{toRedo} depends. Function `redo` is then called on each member of the collection. Only C_{toRedo} and undone commands on which C_{toRedo} redends are redone $\Rightarrow$ the property holds true.

□

6.4.5 Conclusion

The combined undo model presents an user-friendly approach to the selective undo. It combines classic linear undo (extended with the multiple stack support) with its simplicity and selective undo, with powerful feature of undoing any command anywhere in the system.

It also adds a new feature called global linear undo, which allows the user to get the whole document to the state as similar as possible after the execution of selected command. This function operates only over `global_history_buffer` and in fact does not need local history buffers.

In fact, the global linear undo can be the simplest way how to provide undo and redo functionality in the environment with multiple workspaces. If all workspaces share the one history buffer and the local history buffers are omitted, the global linear undo works as the linear undo model.

The implementation of the combined undo model can reuse data structures and several functions from previous algorithms, which eases the coding part of the implementation and prevents unnecessary errors in program.

6.5 Comparison of presented algorithms

Each algorithm aims to provide a slightly different functionality. The big advantage is, that all three of them can be built upon almost the same data structures and therefore if one of them is already implemented, implementation of another means only changing the undo manager. With minor changes, they can even coexist together in one application and the user can decide, which approach suits best to his/her needs.

Extended linear undo is extension of the traditional algorithm and it provides basic undo and redo functionality in environments with multiple workspaces. Its main advantage is simplicity of usage. The users are generally familiar with linear undo and this model acts in the same way. Dependencies among particular commands are automatically found and the user is informed, what is necessary to undo or redo. The model can be implemented in many existing applications, because system of commands and history buffers is very common and usually only slight extensions need to be made.

On the other side, the cascade selective undo model is a simple model, which provides selective undo functionality in environments with multiple workspaces. Selective undo is not widely used in existing applications, but especially in complex environments it can improve performance and speed up the work with applications. Although it is a novel feature, it should not be difficult for the user to learn how to use it.

6.5.1 Behavior difference

Differences among presented algorithms can be illustrated in sample scenario depicted in Figure 6.4.

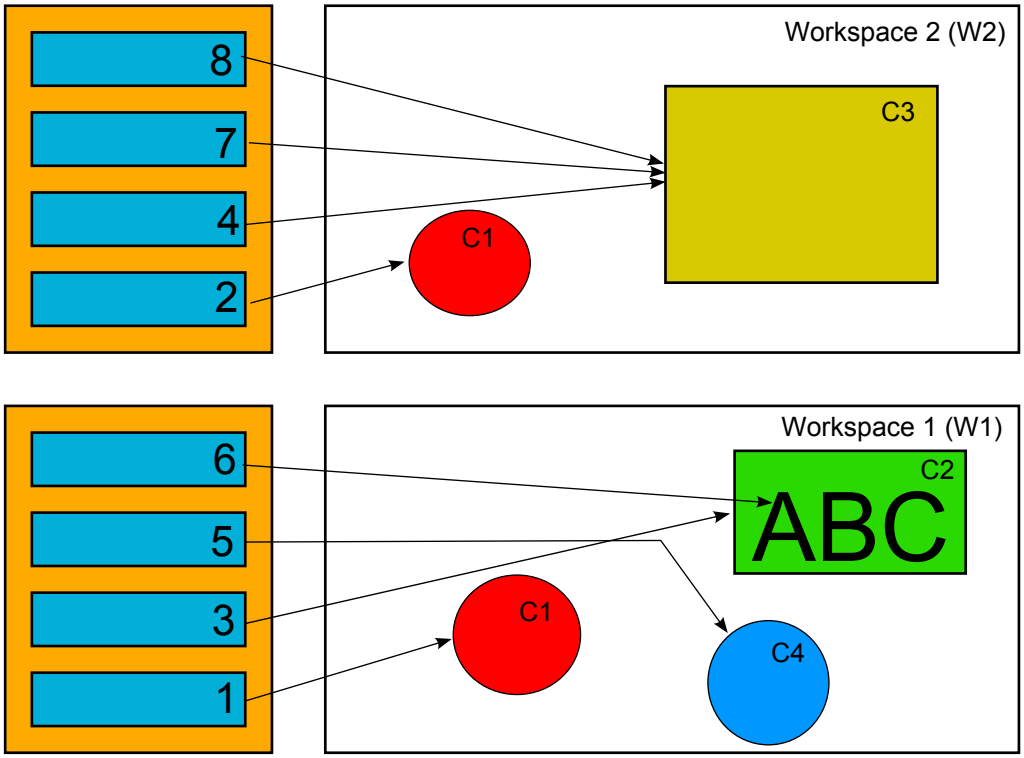


Figure 6.4: Sample situation with two workspaces

The sample scenario was created as follows:

1. Creation of construct C1 (red circle) in workspace 1 (W1), using command 1.
2. Visualization of construct C1 in workspace 2 (W2), using command 2.
3. Creation of construct C2 in W1 using command 3.
4. Creation of construct C3 in W2 using command 4.
5. Creation of construct C4 in W1 using command 5.
6. Modification of construct C2 (addition of text ABC) in W1 using command 6.
7. Modification of construct C3 (resizing) in W2 using command 7.
8. Modification of construct C3 (change of color) in W2 using command 8.

For simplicity, the situation contains only two workspaces and eight commands but it applies to multiple workspaces as well. Now consider the user's intention:

1. The user wants to undo creation of C1 and other constructs should be left untouched.

   - **Extended linear undo model** - It cannot be done, because undo of C1[2] in W1 would cause undoing all commands in both workspaces. The similar action performed in W2 would undo all command in W2, but the C1 in W1 would remain executed and it would need one more undo in W1. This would undo all command in W1 resulting in the same situation.
   - **Cascade selective undo model** - Undo performed in W1 on C1 will cause undo of commands 1 and 2, and therefore no constructs except C1 are affected.
   - **Combined undo model** - Combined undo model can use the selective undo functionality, resulting in the same situation as in case of cascade selective undo model.

2. The user wants to undo creation of construct C4 in W1 and other constructs should be left untouched.

   - **Extended linear undo model** - It cannot be done, because command 6, which is older, is on the same stack and it must be undone prior to command 5. Stack belonging to W2 is left untouched.
   - **Cascade selective undo model** - There is no dependent command, which depends on command 5. Therefore command 5 can be undone and all other commands are left untouched.
   - **Combined undo model** - Combined undo model can use the selective undo functionality, resulting in the same situation as in case of cascade selective undo model.

3. The user wants to undo resizing of construct C3 (command 7), all other commands should remain executed.

   - **Extended linear undo model** - It cannot be done, because command 8 depends on command 7 and thus it have to been also undone.
   - **Cascade selective undo model** - The same behavior as for the extended linear undo model.
   - **Combined undo model** - The same behavior as for the extended linear undo model.

4. The user wants to get the whole document into the state after the execution of command 4.

   - **Extended linear undo model** - The user has to manually undo commands 7 and 5 (it would cause automatic undo of all younger commands in the stacks). The user has to be aware of the global order of commands.

[2]Undo of command in the middle of the stack in the extended linear undo model is generally not possible, because only the command right bellow the top of the stack can be undone. In this case we assume, that selecting C1 to undo would cause sequential undo of commands 6,5,3 and then command 1 would be undone.

- **Cascade selective undo model** - The user has to manually undo commands 8, 7, 6 and 5 or 7 (which undoes also 8 because of the dependency), 6 and 5. The user has to be aware of the global order of commands.
- **Combined undo model** - The user can select command 4 for global linear undo and undo manager will undo younger commands automatically.

This enumeration depicts the most common situations and how particular models can react. Points 1,2 and 4 present possible differences in reactions, point 3 presents a situation in which all 3 models must react similarly.

The most versatile model is the combined undo model, which can use both linear and selective undo, according to actual need.

7. Undo and Redo in DaemonX Project

The project DaemonX has been chosen to serve as a platform for experiments with selective undo models. The reasons and current state of undo and redo in the project are described in following sections. The general aspects of the project are described only briefly and the chapter focuses mainly on the structure of history buffers, dependencies among commands and command design.

Further information about the project itself can be found in project documentation [12](publicly available).

7.1 Project Overview

The DaemonX framework is pluginable tool developed for data and/or process modeling. The word framework is very important - DaemonX as solitaire application is executable program, but it is not very useful. All functionality is provided via various plugins, which use services provided by DaemonX framework.

The main services provided by DaemonX:

- **Integrated environment** - The application DaemonX forms an envelope around all plugins, takes care about particular events, provides methods for saving project state and unites interaction with the user (interface of each plugin looks the same).
- **Tools for plugin development** - The et of abstract classes, which allows plugin programmer to model a modeling language. Also called meta-meta model.
- **Data propagation** - So-called *evolution* allows automatic data propagation between two plugins.
- **Undo/redo support** - The set of abstract classes for commands and integrated undo/redo manager. The manager and the whole undo model is discussed in the following chapters.

7.1.1 Data and Process Modeling

The main purpose of DaemonX framework is data and process modeling. There are many data and process modeling languages and it would be hard to support all of them. Because of this, DaemonX supports so-called *modeling plugins*. A modeling plugin is a binary library, which can be integrated into the framework and it is used to perform actual data or process modeling. It implements an interface specified by DaemonX and uses services provided by the framework. Usually there is one modeling plugin for each modeling language.

Big advantage of this approach is modularity of the application. The user can install only those plugins which he/she really uses. Anybody can develop a new plugin and extend functionality of DaemonX - he/she must only implement the interface specified by DaemonX, which is publicly available.

Modeling Plugin Development

Project DaemonX itself contains several sets of abstract classes, which are used by plugin developers:

- Classes for modeling another modeling language - meta model. This feature speeds up plugin development, because the developer does not have to care about features common to all plugins - storage for constructs, canvas painting, load/save functions etc. The developer has to set properties of construct and create its *controller* and *view*, which is the interface of the construct for the user of the plugin. For both, abstract classes are also available. Controller is a class, which authorizes changes in the construct's properties. View is the interface of construct for the user of the plugin - the developer can choose shape of construct on the canvas, specify where particular properties are visualized and define allowed operations (resize, change of position etc.) and the methods implemented in abstract class catch events and manages most of view-oriented functions.

- Classes for command implementation. Every action which changes a construct in any way should be implemented as a command. An abstract class manages storage of commands in the history buffers, the developer must provide actual "work code" and its inverse for undo.

- Interface for the plugin itself, which must be implemented by the developer. A plugin uses this interface to pass own ribbon toolbar to the framework and the developer can provide own implementation of several framework function (save/load of construct storage).

Developing of plugin is not a complex task, it is mostly providing implementation of an abstract classes. This was an aim - even unskilled developers without any deeper knowledge about internal function of the framework should be able to implement a modeling plugin.

7.1.2 Data Propagation - Evolution

The most valuable feature is so called *evolution*, which allows data binding between two diagrams and derivation of objects from one diagram to another. For this purpose, special plugin (so-called *evolution plugin*) must exist, which describes how data can be mapped and then propagated. The evolution plugin exists for each two modeling languages implemented for DeamonX, for which the evolution is enabled and all evolution plugins are directed - the data propagation is possible only in one direction. If evolution plugin supports propagation from *source language* L1 to *target language* L2, we call the plugin *L1 → L2 evolution pluing*.

Work with evolution is not very complex. Usually, the user creates diagram(s) in one modeling language (for instance UML). Then he/she decides, that it would be useful to use this data model also in, e.g., BPMN [4] diagram. If there is a UML → BPMN evolution plugin present in the framework, he/she can select constructs, which are then propagated to BPMN diagram. This process is called

derivation - it creates constructs in target diagram and setups so-called *evolution references* between the original and derived constructs.

Evolution reference is a key term - it is the only way, how to specify cross-language relationship between two objects in one project. This reference is oriented (one object is a source and the second one is a target) and its existence between two objects means, that if the value in the source object is changed, this change is propagated to the target object. How the propagation is done is specified in evolution plugin - there is no way how to specify some general rules of propagation between two modeling languages - there is no common modeling language and the rules would have to be customized anyway.

In situation that both diagrams (UML and BPMN) already exist, evolution references can be set up by hand in user-friendly designer.

7.1.3 Technical Solution

The basic unit for propagation is *command*. The command in DaemonX matches the definition of the command from in Chapter 3 very well, although it is extended with additional variables.

Evolution plugin contains a predefined set of rules specifying, which values can be propagated and how they can be propagated. All these rules are defined over commands, in fact it is a mapping of commands from source modeling plugin to commands of target modeling plugin.

When a new command C is being executed, it is also passed to *evolution manager*. Evolution manager has a set of evolution references and set of commands. Manager at first checks, whether there is an evolution reference binded to one of command's affected constructs. If so, the manager tries to find a rule, which has similar type of C. If such a rule is found, propagation will occur.

7.2 Environment for Undo

The previous section briefly describes functions and some inner parts of the DaemonX framework. For the purpose of this thesis, the most important part is undo/redo manager and history buffers.

The first aim of the DaemonX framework was testing of evolution concept. It should be an advanced proof of concept for this feature and therefore initial architecture design was evolution-centered. However, other functions were not completely left behind - if the application wants to be successful, it must be at first easy to use.

For these reasons, the undo/redo functionality was a part of initial design, but the proposed architecture was not very sophisticated. Because of evolution, we needed to delimit and abstract operations with constructs to commands, which implies also using commands to undo/redo management. Any other approach would be wasting of resources.

7.2.1 Design of Commands

Atomic command

As stated before, DaemonX provides an abstract class, which is ready for command implementation - `ACommand`. `ACommand` stands for *atomic command*, which is one single piece of action, which can be executed. It is really atomic, so the developer can be sure, that execution cannot be interrupted. Atomic command should perform changes on exactly one construct.

All atomic commands must implement (among others) two methods and two read-only properties:

- `DoExecute` and `CanBeExecuted`
- `DoUndo` and `CanBeUndone`

The `CanBe...` properties perform necessary checks. If they are true it must be guaranteed that subsequent calling of respective `Do...` function will succeed.

Each atomic command carries a key of the construct which affects - this information is later used for building up

For each construct, usually at least 2 atomic commands exist - one for creation and one for deletion. If the construct supports modification of properties, each such a modification is usually done by a separate atomic command - although it is not mandatory and there can be a parametrized atomic command, which can modify several properties at once (but one instance should modify only one particular property).

Manipulation with views is also performed in command based environment. The framework provides general commands for movement and resize of constructs. However the developer may of implement his/her own commands to do this.

Command Group

The atomic command is the smallest piece of action the framework can perform on a construct. The problem is, that in some situations this action is too small. Chapter 3 provides an example of such a situation in Figure 3.2. Deletion of class Vehicle implies deletion of both connections with the subclasses. For the framework, it is not a problem to detect such a situation and solve it. But there is no atomic command which deletes one class and two connections, there are only separate commands for class deletion and for connection deletion. The user does not want to see three separate commands in the history buffer, because for him/her it was one action caused by one click.

For this purpose, DaemonX uses so-called *command groups*. Command group puts several atomic commands together and provides the interface to execute and undone them. In fact, the interface is very similar to command's interface.

When an `Execute` method of command group is called, it executes in sequential order all atomic commands in the group, undo is performed similarly but in reversed order. One command group is `executable` if and only if all atomic commands in the group are executable (property `CanBeExecuted` returns true), similarly for undo. The order of commands in the group is maintained by the creator of the group and it is his/her/its responsibility, that execution order will not bring the document into an unstable state.

There is a special array of commands in the group, called *termination commands*, which are executed and undone in the same order and it is guaranteed, that these commands will be executed or undone after all other commands were executed or undone. The termination commands serve mainly for internal framework purposes (refresh of view etc.) and generally they should not be used by the developer of plugin.

7.2.2 Command Group Tree

Each atomic command carries an information about the key of the affected construct. Command group makes this information public, so it is possible to construct a directed graph of implicit dependencies among command groups. But data propagation process through evolution needs sometimes to explicitly set dependencies among several groups.

For this purpose, *command group tree* has been created. Groups are formed into directed graph - tree. The first added group is selected as a root of the tree. Each group must have exactly one parent (except the root group, which has no parent) and zero to infinity children. Since the graph is a tree, there must not be a cycle. If there is an edge from group A to B, it means that when A is being undone, B should be undone prior to A (A is older than B). The whole tree is executed in layers. The number of the layer, in which the command group resides, is defined as the distance of the groups in layer from the root of the tree plus one. The execution starts with root layer (layer 1), then all command groups in layer 2, then layer 3 etc. Groups in the same layer are executed in order they were added into tree.

Structure of tree is depicted in Figure 7.1.

7.2.3 Undo Management

The whole undo/redo management in DaemonX is controlled by one singleton [10] class called `UndoRedoManager`.

Each command group created by plugin is passed to this class with specification of the workspace in which it was created. The manager accepts only command groups and command group trees - even if only one atomic command is necessary to perform desired action, it must be wrapped into a command group. The manager firstly checks, whether the group is executable and if so, the group is executed and stored in manager's internal data structures.

When a command group tree is passed to the undo/redo manager, it at first traverses through the tree and updates keys of affected constructs of each command group according to position of the group in the tree. Manager itself does not maintain any database of dependencies, all of them are stored as keys of affected constructs in particular atomic commands or command groups. When update is finished, it executes groups layer after layer starting with the root (layer 1). Each executed group is then stored in the right history buffer.

Internal data structures of manager are so-called *command stacks*. Because undo/redo support was not a main task in DaemonX, the first idea was to use a global linear undo - all commands, even from different plugins, are mixed in one history buffer. This approach is not novel, it was already used for project

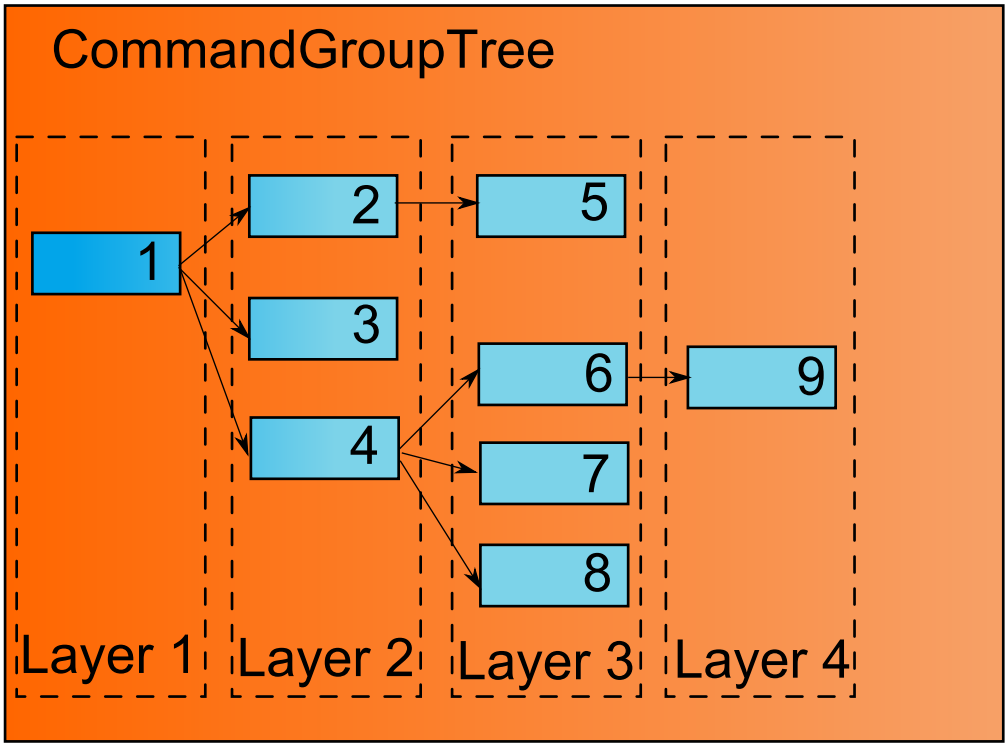


Figure 7.1: Command group tree

XCase [3]. It is simple to implement, but after testing we realized, that it is not very user-friendly. When undo is performed, the application jumps from one workspace to another, and it is hard to control, which action will be undone in the next step. Therefore, there is a separate history buffer for each workspace.

The overview of this situation, together with the structure of command group, is depicted in Figure 7.2.

Extended Linear Undo

The first implemented algorithm was extended linear undo. It was developed during initial development and proven to be useful feature.

The commands were managed stored in stacks. Each stack maintained its own top of the stack, in other aspects it was just a collection of commands.

For dependency search, one global command buffer was used. This buffer contained reference to each command in the document, ordered from the oldest to the youngest.

When undo or redo operation was performed and the dependent command is found, a dialog is shown to the user, with specification of dependent commands. Each entry contains a checkbox - the user can select, which commands will be really undone or redone. Dialog assures by automatic checking of particular checkboxes, that there will not be an undone command between two executed commands and vice versa. However, if the user unchecks any command to undo, it may bring the project into unstable state. If all entries remain checked, it is guaranteed that the document will not be in unstable state the desired operation. A sample dialog is shown in Figure 7.3.

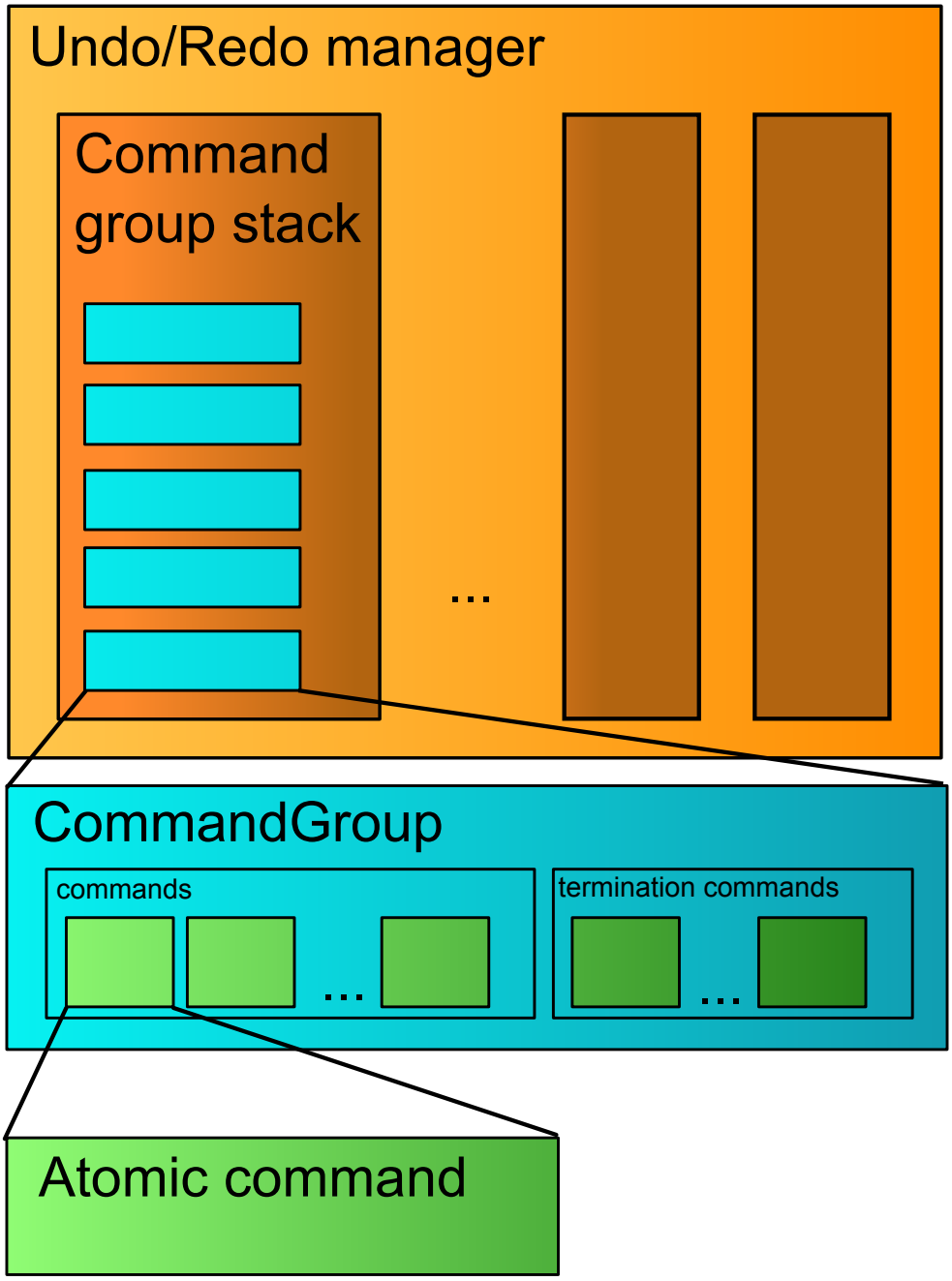


Figure 7.2: Undo/Redo manager with several history stacks

The purpose of this dialog is mainly to inform the user, that also other commands are being undone, but it allowed the user also to solve the situation, when unnecessary command was being undone.

The dialog is still present in both following models.

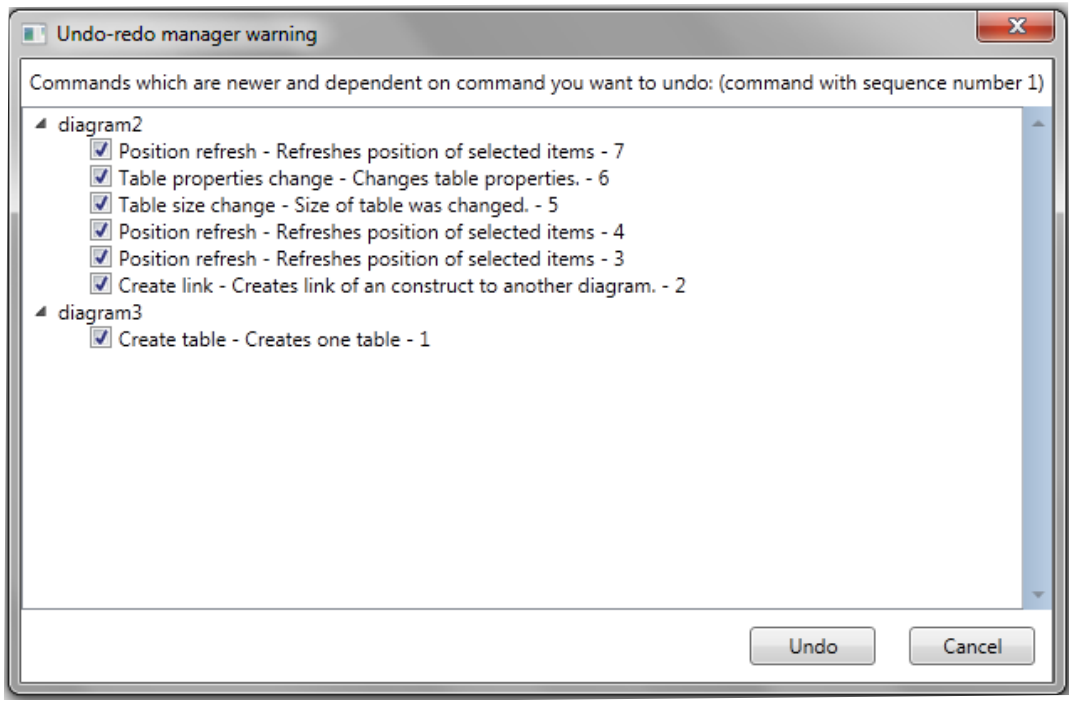


Figure 7.3: Dialog shown by undo/redo manager, when a dependent commands were found

Cascade Selective Undo

The second implemented algorithm was cascade selective undo model. This model was also useful, but impossibility of undoing large chunks of commands was a limitation.

The implementation follows the analysis done in Chapter 6. Command stacks are just collection of commands, which can return command based on its position in the stack or based on its key.

All important work is done in undo/redo manager, which controls the stacks. When undo or redo is performed, it searches for dependencies and undoes or redoes commands returned by the search method.

The dialog from the previous section is displayed when at least one dependent command is found. Its behavior is slightly changed - the user is able to select any combination of checkboxes.

Interface to the user was also changed - undo and redo button were removed and the user selects the command to undo or redo by clicking into the list of commands on the particular command.

Combined Undo

The third and last implemented algorithm is the combined undo model.

The implementation mostly follows analysis done in Chapter 6 with one exception. Each stack manages its own virtual top of the stack and when a new command is executed, all commands above the virtual top in current stack are discarded. It is expected, that if the user has undone a command, it has been undone on purpose and therefore when a new command is executed, these commands can be safely discarded from the stack.

Because the model supports three methods of undo, the user has to be able to select which one he/she wants to use. The command stack interface was extended with the context menu, which reacts on right mouse button (RMB). When the user clicks RMB on the command he/she can select from the menu, whether he/she wants to use linear undo/redo, selective undo/redo or global linear undo. Selecting linear undo (redo) on the command in the middle of the stack causes linear undo (redo) on all executed (undone) commands above (below). Buttons "undo" and "redo" are present and they cause linear undo and of command right below or at the virtual stack top.

The example of the user interface is depicted in Figure 7.4. We can see eight commands in the stack, four are undone (green) and four are executed (blue). At the top of the stack, there two buttons with small arrows - the undo/redo button, which causes linear undo/redo. In the middle of the picture, we can see the context, which is displayed after right click on the particular command and which offers three possible ways, how to undo the command. The commands carry a simple information about their purpose (e.g. "Create UML initial state"). However, this information is only static and does not inform the user, which particular construct is affected.

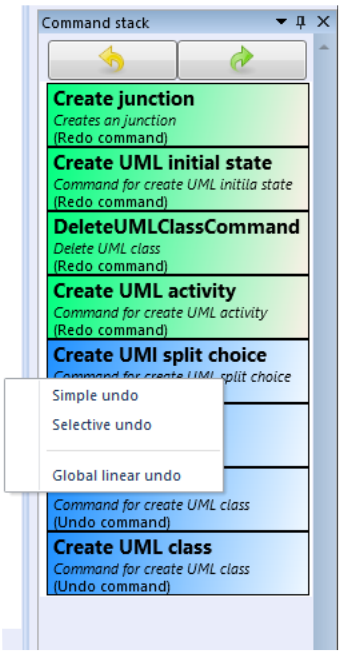


Figure 7.4: The user interface for the combined undo model.

Data structures were reused from the previous models and only the undo/redo manager was changed.

7.3 Results

All three models were tested during usage of DaemonX. However, lack or real-world users of DaemonX project limits results to impression of developers during testing of various plugins.

7.3.1 Extended Linear Undo Model

The extended linear undo model was present during initial development and therefore it was extensively tested. The general impression was good, the model was easy to use and in case of one workspace or several independent workspaces, it acted like linear undo. The need of undoing dependent commands (and all commands above them in the same stack) were sometimes annoying and because of this, the user in some cases did not use the undo/redo operations, but he/she fixed the problem in an other way (manual deletion of construct,...).

Results of the model were easy to predict, but only on one stack. When undo or redo operation affected several stacks, it was hard to determine, in what state the document will be. This problem could be fixed by developing even more extended version of commands, which could inform the user about actual changes done to the constructs. However this change would mean reprogramming several hundreds of commands and therefore it has been selected as potential future work.

7.3.2 Cascade Selective Undo Model

The cascade selective undo model brought a possibility of undoing any command, but its limitation was undoing bigger chunks of commands at once.

This feature was not present and if the user wanted to undo fifteen independent commands, he/she had to fifteen times select the particular command to undo. This feature was really wanted by users and therefore it was the reason, why cascade selective undo model was not so popular as plain extended linear undo model.

The selective undo feature itself was mainly appreciated by the users and they proposed a wish to combine both models, which resulted in combined undo model.

7.3.3 Combined Undo Model

Combined undo model takes the best from both previous models and negates their disadvantages.

When the user wants to undo only single command in the middle of the stack, he/she can use a selective undo. If there is a need to undo big a chunk of commands, linear undo or global linear undo can be used.

Global linear undo feature was really appreciated by the users, which shows that linear approach is still probably more natural than selective.

The used user interface - undo/redo buttons and list of commands with context menu - is generally sufficient, but its improvement can be a part of possible future work.

8. Conclusion and Future Work

The aim of the thesis has been designing and providing an algorithm or model, which would be able to manage selective undo and redo operations in environments with multiple workspaces, where particular actions may depend on each other.

This thesis starts with a brief summary of several different approaches to undo/redo and it provides a comparison of 6 real-world undo models with emphasis on non-linear undo models. This experience results in definition and analysis of selective undo and connected issues. For each of them several potential solutions are discussed and their advantages and disadvantages are depicted.

The key part of thesis is a proposal of three undo models - extended linear undo model, cascade selective undo model and combined undo model.

The first one extends linear undo model for environments with multiple workspaces and it proved the concept of dependency searching to be useful and effective. The model can be used in applications, where simple global linear undo is a limitation for comfort use.

The second one provides universal implementation of selective undo model and verifies particular solutions of issues presented during the analysis. The algorithm performs well also in simple environments with only one stack and it can be used in wide range of applications.

The third model combines both approaches together and adds function of global linear undo. It is the most sophisticated model presented in this thesis and it allows the user to choose between linear and selective undo according to actual needs. It discusses limitations of both approaches in one model and one application and it proves.

All three models are easy to implement, because they are built upon a common system of commands and history buffers, which are widely used in today applications. The ease of the implementation has been proven trough DaemonX framework, which served with its complex system of workspaces as a laboratory for comparison of the models. The final implementation of combined undo model extended the functionality of the framework itself.

8.1 Future Work

The general aim of the thesis has been fulfilled, but not all issues of undo and redo operation are solved and there are many ways how to improve existing or to create new models.

The big challenge is design of a selective undo model, which would search dependencies among commands based on the multi-criterial basis. Presented models use only keys of affected constructs to decide, whether there is a dependency between two commands. Future models may use also other criteria - for instance type of command(creation, modification, deletion, move,...) can be taken into account. This approach could reduce the amount of commands selected as dependent in one undo or redo operation. It implies revisiting *modify already modified* issue by careful selecting criteria, which makes the command dependent (movement commands depends on creation, but not on modification etc.).

The other challenging task is to revisit the user interface of undo manager. Almost all current implementations use two buttons and a stack of actions, but in case of selective undo, also different approaches can be meaningful. Commands in the stack can be ordered not by the age but by its state (executed, undone), stack can visualize only commands for the focused construct. Also the history trees, like in US&R model [18], can used for having several versions of one construct in the one stack.

Bibliography

[1] Bsd system calls manual - undelete(2). `http://www.unix.com/man-page/freebsd/2/undelete/`.

[2] Scalable vector graphics (svg). `http://www.w3.org/Graphics/SVG/`.

[3] Xcase - a tool for xml data modeling. `http://www.ksi.mff.cuni.cz/xcase/index.html`.

[4] Business Process Modeling Notation (BPMN) Version 1.2. Technical report, January 2009.

[5] James E. Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.*, 6:1–19, January 1984.

[6] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1:269–294, September 1994.

[7] Arthur W. Burks and Alice R. Burks. First general-purpose electronic computer. *IEEE Ann. Hist. Comput.*, 3:310–389, October 1981.

[8] Aaron G. Cass, Chris S. T. Fernandes, and Andrew Polidore. An empirical evaluation of undo mechanisms. In *Proceedings of the 4th Nordic conference on Human-computer interaction: changing roles*, NordiCHI '06, pages 19–27, New York, NY, USA, 2006. ACM.

[9] Alan Dix. Moving between contexts. In *DESIGN, SPECIFICATION AND VERIFICATION OF INTERACTIVE SYSTEMS '95*, pages 149–173. Springer, 1995.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[11] O. M. G. Group. Omg unified modeling language, infrastructure, 2009. Version 2.2.

[12] Karel Jakubec, Marek Polák, Vladimír Kudelas, Martin Chytil, and Peter Piják. *DaemonX framework manual*, jun 2011.

[13] George B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8:50–87, January 1986.

[14] Roberta Mancini, Alan J. Dix, and Stefano Levialdi. Dealing with undo. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, INTERACT '97, pages 703–705, London, UK, UK, 1997. Chapman & Hall, Ltd.

[15] Lance A. Miller and John C. Thomas. Behavioral issues in the use of interactive systems. In *Interactive Systems, Proceedings, 6th Informatik Symposium*, pages 193–216, London, UK, 1977. Springer-Verlag.

[16] Trygve Reenskaug. The model-view-controller (mvc) its past and present, 2003.

[17] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9:309–361, December 2002.

[18] Jeffrey Scott Vitter. Us&r: A new framework for redoing (extended abstract). *SIGSOFT Softw. Eng. Notes*, 9:168–176, April 1984.

[19] Yiya Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, 1988.

Appendix A

Content of the CD

The attached CD contains:

- `/thesis/thesis.pdf`
- `/DaemonX`
  - `/bin` - Compiled release version of DaemonX with necessary plugins installed, ready to run.
  - `/doc` - Documentation to the DaemonX project.
  - `/src` - Source codes of DaemonX project.

Appendix B

Combined Undo Model Manual

The usage of undo/redo in DaemonX is not complex. There are two areas, where the undo/redo can be invoked, both are marked in Figure 8.1 with black arrows.

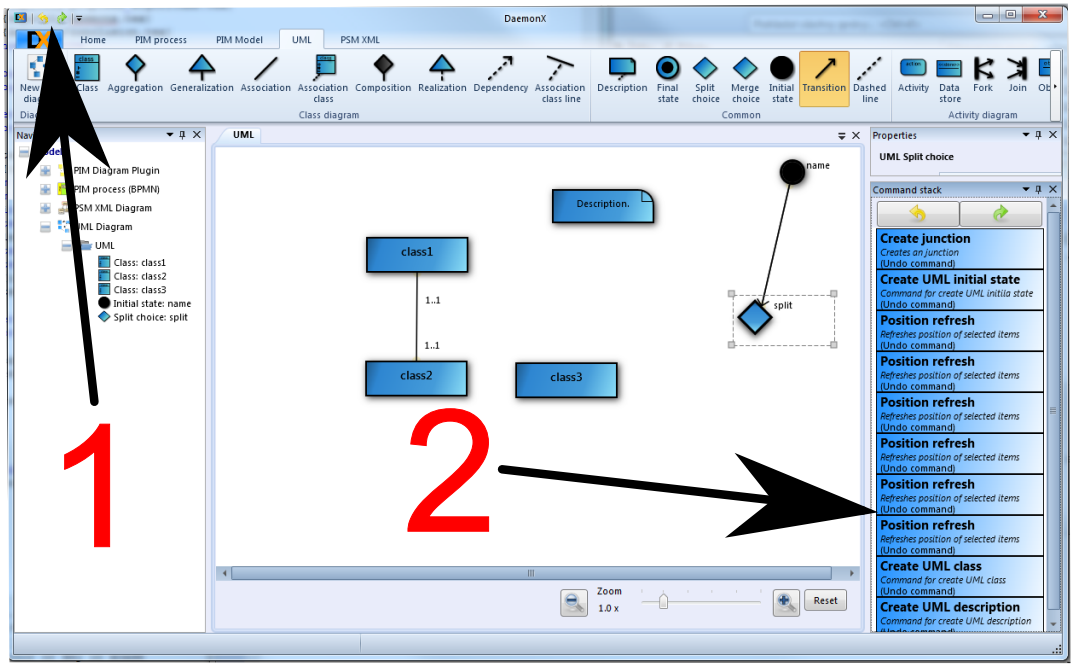


Figure 8.1: The areas, where undo/redo can be controlled.

The first one (marked with red "one") is located at the top of the main window and contains two small undo/redo buttons. Both will trigger only linear undo/redo in the document.

The second one (marked with red "two") is in the floating window with the title "Command stack". The window is located by default in the bottom right corner of the main window, but the user can move it freely. The two buttons at the top of the command stack window offer the same functionality as the undo/redo buttons at the top of the main window.

Twe blue or green rectangles it the command stack window represent commands. The green ones are undone, the blue ones are executed. The most recent (the youngest) command is always on the top of the stack - it is the first command in the window. The amount of commands displayed in the command stack window can be controlled via DaemonX general settings [12].

To trigger selective undo or the global linear undo, the user has to click with the right mouse button on the particular command in the command stack. The context menu, which is depicted in figure 8.2, is displayed.

Then, the user can choose three types of undo/redo:

- Simple undo/redo - The linear approach.
- Selective undo/redo - The selective approach.

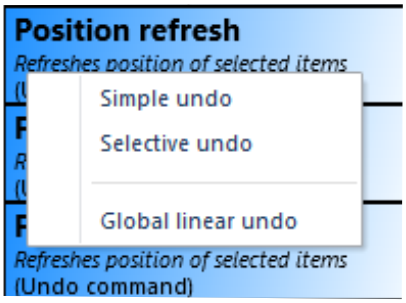


Figure 8.2: Context menu - undo the command.

- Global linear undo

After clicking on one option, the chosen method of undo/redo will be performed and the selected command will be undone/redone. But before the actual undo/redo of the selected command, the undo/redo manager will search through the global stack for dependent commands. If at least one is found, the dialog depicted in Figure 8.3 appears.

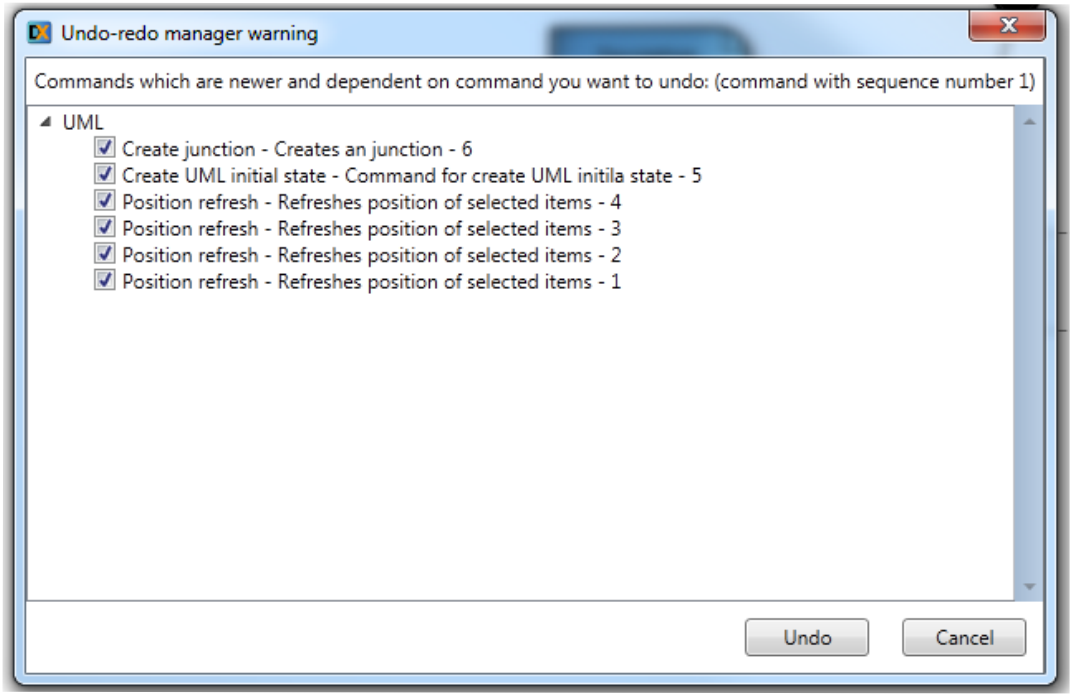


Figure 8.3: Commands which are going to be undone/redone.

The dialog summarizes the actions to be taken - which commands will be undone/redone. The user can take a command out the list by checking it out. However, checking a dependent command out may cause an unstable state of the document and it should be used for the development purposes only.